NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DETERMINE NETWORK SURVIVABILITY USING HEURISTIC MODELS

by

Eng Hong Chua

March 2003

Thesis Advisor:	Geoffrey Xie
Second Reader:	Bert Lundy

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Title (Mix case letters) Determine Network Survivability Using Heuristic Models	**5. FUNDING NUMBERS**
6. AUTHOR(S) Eng Hong Chua	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

Contemporary large-scale networked systems have improved the efficiency and effectiveness of our way of life. However, such benefit is accompanied by elevated risks of intrusion and compromises. Incorporating survivability capabilities into systems is one of the ways to mitigate these risks.

The Server Agent-based Active network Management (SAAM) project was initiated as part of the next generation Internet project to address the increasing multi-media Internet service demands. Its objective is to provide a consistent and dedicated quality of service to the users. SAAM monitors the network traffic conditions in a region and responds to routing requests from the routers in that region with optimal routes. Mobility has been incorporated to SAAM server to prevent a single point of failure from bringing down the entire SAAM server and its service.

With mobility, it is very important to select a good SAAM server locality from the client's point of view. The choice of the server must be a node where connection to the client is most survivable. In order to do that, a general metric is defined to measure the connection survivability of each of the potential server hosts.

However, due to the complexity of the network, the computation of the metric becomes very complex too. This thesis develops heuristic solutions of polynomial complexity to find the hosting server node. In doing so, it minimizes the time and computer power required.

14. SUBJECT TERMS Fault Tolerance, Network, Reliability, Survivability, Server Placement			**15. NUMBER OF PAGES** 115
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approval for public release; distribution is unlimited.

DETERMINE NETWORK SURVIVABILITY USING HEURISTIC MODELS

Eng Hong Chua
Defence Science & Technology Agency (Singapore)
B.Eng.(Honors) Nanyang Technological University (Singapore), 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2003

Author:	Eng Hong Chua
Approved by:	Geoffrey Xie Thesis Advisor
	Bert Lundy Second Reader
	Peter Denning Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Contemporary large-scale networked systems have improved the efficiency and effectiveness of our way of life. However, such benefit is accompanied by elevated risks of intrusion and compromises. Incorporating survivability capabilities into systems is one of the ways to mitigate these risks.

The Server Agent-based Active network Management (SAAM) project was initiated as part of the next generation Internet project to address the increasing multi-media Internet service demands. Its objective is to provide a consistent and dedicated quality of service to the users. SAAM monitors the network traffic conditions in a region and responds to routing requests from the routers in that region with optimal routes. Mobility has been incorporated to SAAM server to prevent a single point of failure from bringing down the entire SAAM server and its service.

With mobility, it is very important to select a good SAAM server locality from the client's point of view. The choice of the server must be a node where connection to the client is most survivable. In order to do that, a general metric is defined to measure the connection survivability of each of the potential server hosts.

However, due to the complexity of the network, the computation of the metric becomes very complex too. This thesis develops heuristic solutions of polynomial complexity to find the hosting server node. In doing so, it minimizes the time and computer power required.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to extend my gratitude to Professor Geoffrey Xie for his patience and willingness to teach and advise me with my research. His guidance and enthusiasm has helped to carry this project to completion.

I would also like to acknowledge the contribution and help that Lt. Baris, Aktop had rendered in various part of this project

I thank the Defence Science & Technology agency of Singapore for this opportunity that they have given me to learn and further my education in a conducive environment like this – The Naval Postgraduate School.

I would also like to thank my parents for their love and sacrifice, without whom I would not have the opportunity that I am enjoying today.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND OVERVIEW

A. BACKGROUND

As organizations try to keep pace with today's information technology, they have their distributed network systems and services connected to the Internet. However, accompanied closely with the benefits of networking is a long list of security issues. Most of these systems are operating at a very high risk of being susceptible to malicious attacks. Furthermore, it is known that no amount of system hardening can assure that such systems, when connected to an unbounded network such as the Internet, are impenetrable to attacks. The only way is to minimize these vulnerabilities.

Since the risk of compromises and intrusions always exists, and no amount of time and money can totally eliminate them, the effort should be focused more on the survivability of the system. The objective of survivability is to ensure that a system can continue to deliver essential services even in the presence of attacks and intrusions.

Survivability is a fairly new concept. This concept started off at late 1995 or early 1996 [LIP00], when large-scale networked systems increase the vulnerability to intrusion, compromises and failures of systems. As an emerging discipline, survivability builds on related fields of study such as security, fault tolerance, safety, reliability, reuse, verification, and testing. On the other hand, survivability encompasses unique concepts and principles.

Since the introduction of the survivability concept, there is little implementation of survivability capabilities into systems. Even if there is an intention of incorporating survivability, it is often taken as a separate thread of the project development cycle, and as a result, survivability is often considered as an add-on property. Survivability should be integrated and treated on par with other system properties in order to develop systems that are able to withstand failures and compromises.

Server Agent-based Active network Management (SAAM) is a network management system, which is developed as part of the next generation Internet project to address the issue of Quality of Service (QoS) for Internet users. SAAM constantly observes the network and determines the best route for each in-coming routing request.

With such a crucial task at hand, it is very important that the SAAM service remains operative at all times. In other words, the system needs to be extremely survivable.

A few graduate students of the Naval Postgraduate School have implemented fault tolerance features into the SAAM system. One of them, Scott Margulis, had taken a major step in incorporating this effort. He brought in the concept of mobile agent software. Margulis's thesis [Mar01] looks into the mobility of SAAM server application, and implements Margulis AGent-based Mobile Applications (MAGMA) on the SAAM. MAGMA enabled the SAAM server to migrate manually from one host to the other, or when there is a disruption to of the SAAM service in the network due to the failure of the hosting node.

However, Margulis's thesis does not address the problem of "where and when" to move the SAAM service to a new host. In one of the "Network Design and Programming" courses, Baris Aktop, Ekrem Serin and Selcuk Ozturk worked on this problem as their class project. In their project report [AKT02], an ad hoc model is developed to move the service around the SAAM region randomly. A metric, Fit to Serve Index (FSI), is defined for each possible host, which changes according to a set of rules. The SAAM service is always relocated to one of the top five hosts with higher FSI values.

From the survivability point of view, the hosting router must be the one at the most optimal locality of the network topology in order to efficiently service all its clients. However, the required computation to find this node in the network is very complex, especially when the network is large and involves large number of nodes. A heuristic model with polynomial complexity should be developed to determine the optimal node.

B. APPROACH

SAAM is an on-going research project. Since its commencement in May 1998, it has evolved greatly to the current baseline with the capability to support up to 40 lightweight routers, with fault-tolerance features.

Under its current configuration, the SAAM server has the ability to relocate from one host to another, through MAGMA. The migration, however, is mostly carried out manually. Manual migration limits the mobility of the SAAM server agent. For continuity of its service, the SAAM server needs to be able to migrate automatically when an attack takes place on the host or even before any attack occurs to confound the adversary.

Incorporating the intelligence to select the most optimal node to host the server enhances not only the efficiency of the system but also its survivability. A poor choice will degrade the SAAM service, as it may lead to poor reliability for the connections between the server agent and its clients (i.e., routers). It may also result in excessive relocation of the service.

Therefore, it is important to have an appropriate metric to guide the selection of a server host. The metric should measure the suitability of a host based on one or more survivability objectives. To simplify the problem, this thesis considers the survivability of the SAAM service with respect to one client router. It should be straightforward to extend the per-router survivability metric into a network wide metric, e.g., using a weighted average. The survivability objective of this work is to maximize the reliability of connection between the server host and the client router.

The problem of finding the optimal node to host a service has been studied by other people without considering survivability over the past few years. [CAM02] and [KAR01] are some examples showing how to determine the placement of server applications in the Internet. The optimal location is identified and determined based on the customers.

C. SCOPE

Baris Aktop, a graduate student, and the author of this thesis, under the supervision of their thesis advisor, Professor Geoffrey Xie, look into ways to further improve the mobile SAAM server agent. The objective is to establish a rigorous method to determine when and where to move the agent. The work involves formulating a quality of connectivity metric and developing efficient heuristic models based on the metric. Particularly this thesis implements the heuristic models based on the Java-based Graph Algorithm Platform (JGAP) developed by Ding-Yi Chen of Institute of Information Science, Taiwan. The JGAP source code is freely available over the Internet at website http://im.ncnu.edu.tw/~tsai/definite/JGAP/JGAP.html. This thesis also develops evaluation criteria and conducts tests to compare and contrast the accuracy of each of the heuristic models. Random graphs are created for the tests.

D. OVERVIEW OF THE THESIS

This thesis is structured into the following chapters.

Chapter II: Survivability Concept and Other Definitions. Cover the concept of survivability and the associated properties. Describe the Survivability Network Analysis technique. Discuss the differences between survivability and other common engineering terms, such as fault tolerance, reliability and security.

Chapter III: Connectivity Based Survivability Metric. Describe a survivability metric for a potential server host based on the quality of connection between the host and the client. Define the basic terms of the graph theory related to the reliability of network connections.

Chapter IV: Heuristic Models for Rating Quality of Connectivity. Describe each of the heuristic models, which can be used to rate the survivability of different server hosts without having to compute the actual values of the survivability metric.

Chapter V: Experiments and Analyses. Describe the details of the evaluation criteria developed and tests carried out by this thesis. It covers the design of the test program, the test results, as well as the analyses of the results.

Chapter VI: Conclusion and Future Work. Summarize the results from this thesis and suggest areas for future work.

II. SURVIVABILITY CONCEPT AND OTHER DEFINITIONS

A. INTRODUCTION

What is survivability, and why is survivability so important? There are already terms like fault tolerance, reliability, redundancy and security. Are not these sufficient to address the intended system requirement? Why should we incorporate survivability into our systems? Is the additional effort and time spent worthwhile? The answers to these questions are given below.

Contemporary large-scale networked systems have brought together a tremendous revolution in the way we communicate, share information, perform business, and live. The Internet has brought people closer together and helped to make corporations and businesses more efficient and more effective. However, all these benefits are accompanied by elevated risks of intrusion and compromise.

According to the "Computer Crime and Security Survey" [POW02] conducted by the Computer Security Institute (CSI) and the computer intrusion squad of the San Francisco Federal Bureau of Investigation (FBI), computer crimes and other information security breaches continue to increase. Malicious attacks to bring down the networked services have almost doubled from 1997 to 2002. The highest reported monetary loss due to denial of service attack is $50 million.

As systems subjected to attacks are prevalent and inevitable, properties like fault tolerance, reliability that mask the systems against failure; or even security that shields the systems against attacks are not sufficient. These systems need to be survivable in this type of harsh environment. Survivability ensures that the system continue to deliver the essential services, despite of component failures and attacks. More definitions and details of these properties will be described in the following sections.

B. CONCEPT OF SURVIVABILITY

According to [ELL99], survivability is defined as the capability of a system to complete its missions in a timely manner, in the presence of attacks, failures, or accidents. The term "system" refers to any generic system, which can be operating system, dedicated applications system, network interface card, personal computer, network, or system of systems.

The term "mission" is defined as a set of critical specifications of tasks that must be met. These specifications can be in the aspect of functionalities, performance, security, reliability, real-time responsiveness and correctness. Attacks, which are potentially damaging events caused by a well-trained adversary, can include intrusions, probes, and denial of services. Failure and accident are potential destructive events caused by deficiencies in the system or external components, which the system depends on; or natural disasters. These events may exist independently or in consequences of one another. However, in the context of survivability, the concern is not on the damaging events, rather the focus will be on the impact of the events under these adverse conditions, specifically whether the system will still be able to fulfill its mission. As long as the system is able to meet its mission's specifications, it is considered survivable, regardless of any component failure it may have.

At times, services catered under adverse conditions may be degraded or different. This can be due to the failure of some components, such that the workload of these components is re-distributed among the others, potentially overloading them. The system can still be considered survivable, if it can accomplish its missions. In some other cases, it is considered as not fully survivable. Obviously, system survivability is not a binary property of the system but a property that gives a measure of the extent to which it is able to satisfy the critical specifications as defined by the mission.

Naturally, under normal or adverse conditions, a system that is able to meet more of its mission specifications is considered more survivable than one that is not. However, different weights may be given to the specifications. One system failing to satisfy its main specification may have a lower survivable value than one not meeting a number of minor specifications. In any case, survivability is a measure of a system's ability to meet its mission specification, rather than the failure of the system or its components.

A survivable system can also be seen as a system that can repair itself or degrade gracefully to preserve as much critical functionality as possible in the face of attacks and failures. [DARPA98] Since the loss or degradation of components is inevitable, it is essential that such systems continue to function as much as possible when their subsystems fail.

1. Properties of Survivable Systems

Since the objective of survivability is to preserve the essential services, it is imperative to identify these main services, and the properties the system needs to have in order to support these services against failure and attacks. [LIN99] states that survivable systems must exhibit four properties against attacks, namely resistance, recognition, recovery, and lastly adaptation and evolution. These four properties are described in Table 1.

Resistance is the capability of a system to repel attacks. Recognition is the system's capability to detect attacks when they occur and evaluate the extent of damage caused. Recovery is the capability to maintain essential services and assets during attacks, and limit the extent of damage and restore full services following the attack. Adaptation and evolution is the capability to improve the system survivability based on the knowledge gained from previous intrusions.

Key Property	Description	Example Strategies
Resistance to attacks	Strategies for repelling attacks.	Authentication Access control Encryption Message filtering System diversification
Recognition of attacks and extent of damage	Strategies for detecting attacks (including intrusions) and understanding the current state of the system, including evaluating the extent of the damage.	Intrusion detection Integrity checking
Recovery of full and essential services after attack.	Strategies for restoring compromised information or functionality limiting the extent of damage, maintaining or, if necessary, restoring essential services within the time constraints of the mission, restoring full service as conditions permit.	Redundant components Data replication, System backup and restoration Contingency planning
Adaptation and evolution to reduce effectiveness of future attacks	Strategies for improving system survivability based on knowledge gained from intrusions.	New intrusion recognition patterns

Table 1. Some Strategies for Improving System Survivability

In addition to the four properties, survivability should be proactive as well as reactive [WEL99]. The system must have the capability of avoiding brittle configuration. This is illustrated by an example shown in Figure 1. In Figure 1a, it shows a system with two essential services, service A and B. Each service is made highly available using 3 replicas each. A series of attacks bring down 2 replicas of service B (Figure 1b). This made service B vulnerable to failures. If attempts to bring service B's replicas back to operation fail, reconfiguring of one of Service A's replica to support service B is required to increase the overall system survivability (Figure 1c).

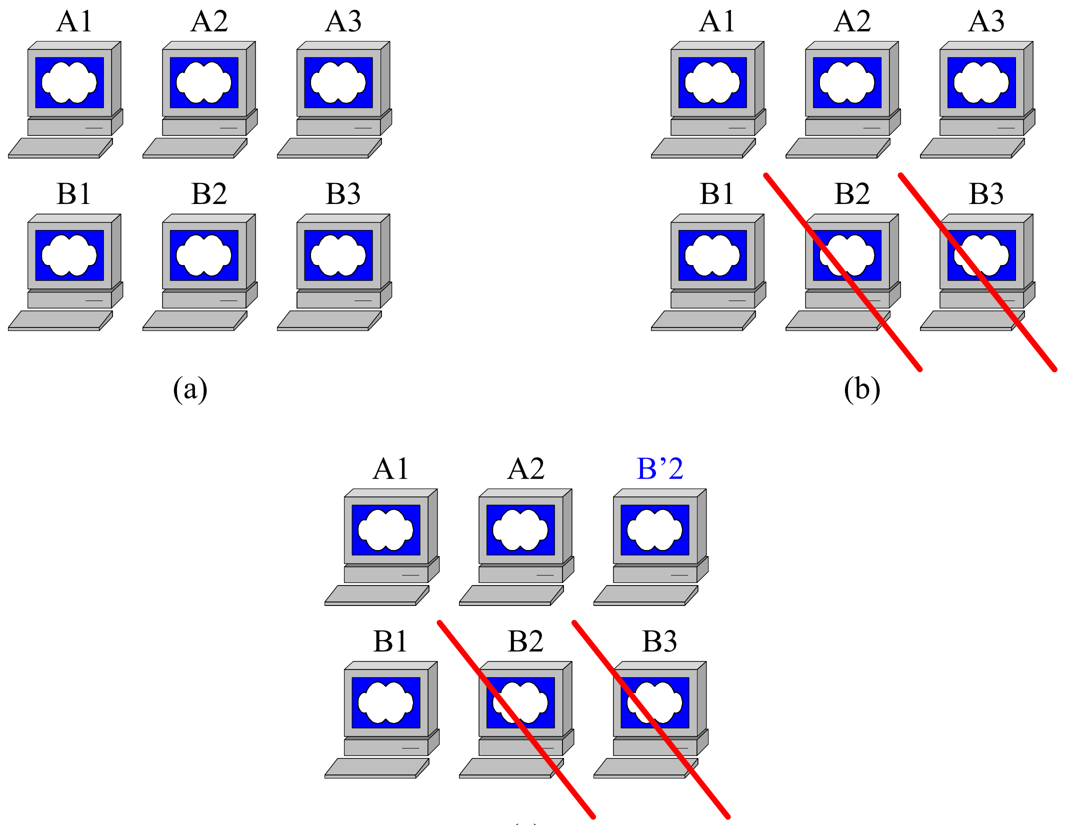

Figure 1. Avoidance of brittle configuration

2. Survivable Network Analysis

Survivable Network Analysis (SNA) was developed by the Software Engineering Institute's Computer Emergency Response Team (CERT) Coordination Center to provide systematic assessment of survivability properties of proposed systems, existing systems, and modifications of existing system. [ELL99]. The analysis is carried out at the architecture level, and its method involves going through a series of joint working sessions, culminating in a briefing on findings and recommendations. Figure 2 depicts the four-step SNA process.

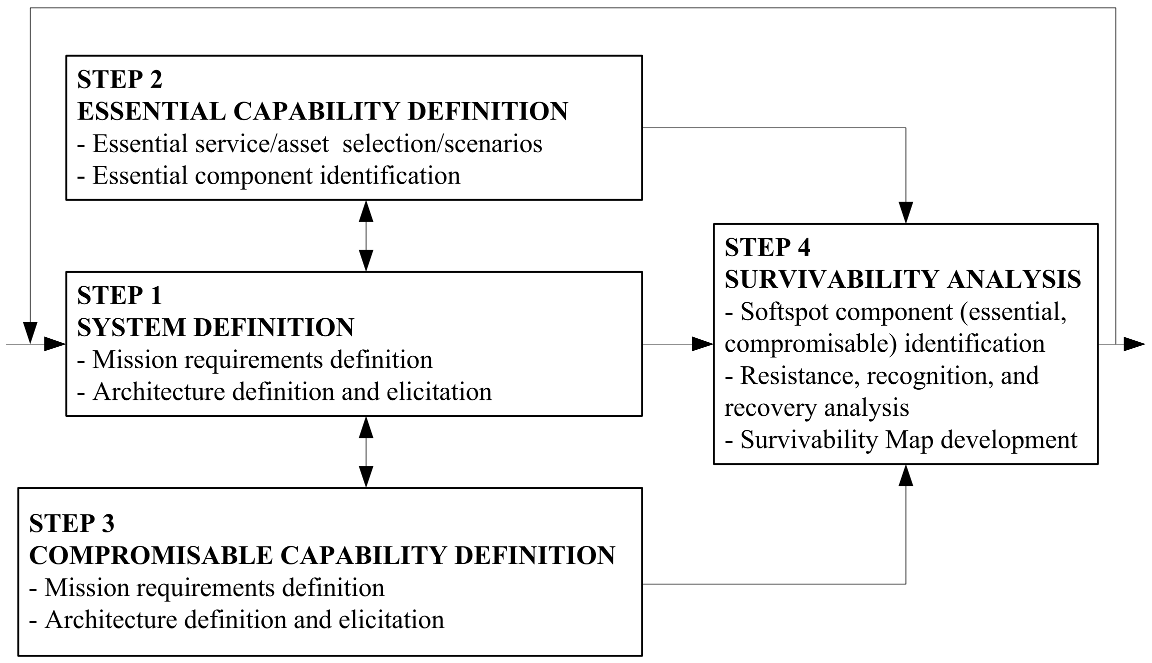
STEP 2
ESSENTIAL CAPABILITY DEFINITION
- Essential service/asset selection/scenarios
- Essential component identification
STEP 1
SYSTEM DEFINITION
- Mission requirements definition
- Architecture definition and elicitation
STEP 3
COMPROMISABLE CAPABILITY DEFINITION
- Mission requirements definition
- Architecture definition and elicitation
STEP 4
SURVIVABILITY ANALYSIS
- Softspot component (essential, compromisable) identification
- Resistance, recognition, and recovery analysis
- Survivability Map development

Figure 2. Survivable Network Analysis Method

In Step 1, System Definition focuses on understanding of the system mission objectives and primary functional requirement. It also looks into the structure and properties of the system architecture and the risks the system may have to contend with in the operational environment.

In Step 2, Essential Capability Definition involves the identification of the essential service and assets of the system based on the system mission objectives and the consequence of failure. Essential services and assets highlight the essential components of the systems that must be always survivable in order not to disrupt the services.

In Step 3, Compromisable Capability Definition selects a set of probable intrusions based on the system's operating environment. With intrusion usage scenarios defined, compromisable components, where intrusion can successfully penetrate and damage, are identified.

In Step 4, Survivability Analysis, soft spot components of the architecture, which are essential and compromisable, are identified and analyzed in term of their capability to resist, recognize and recover from intrusions.

C. FAULT TOLERANCE

Fault Tolerance is defined as the use of multiple sub-systems or components in parallel to provide redundancy, so that the system can continue operating when sub-systems or components fail. Standard terminologies used for fault tolerance are failure, error, fault and error latency. Each system has specified behavior. A failure is a deviation of the actual behavior from the specified behavior. An error is a system state that is likely to lead to a failure. A fault is a physical defect, imperfection, or flaw that occurs within some hardware or software components. It can be caused by specification mistakes, implementation mistakes or component defects or external disturbance such as environmental effects.

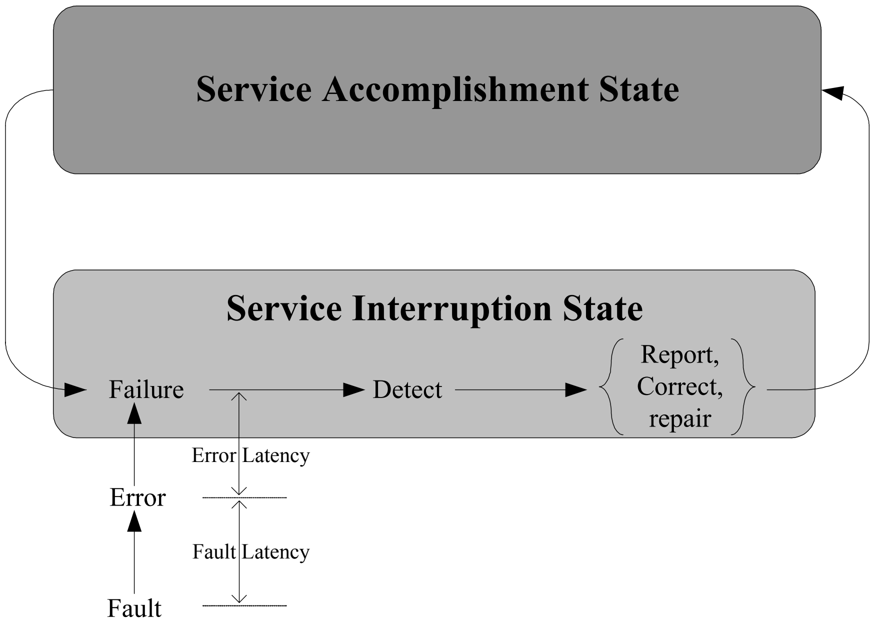

Figure 3. Service States of Fault Handling

When the system operates according to its specified behavior, it is said to be in the service accomplishment state, as shown in Figure 3. However, occasionally, a fault creates an error causing the system to fail. In that case, the system enters the service interruption state. The failure needs to be detected, reported and corrected before the system can return to a service accomplishment state. The time between the occurrence of a fault and the appearance of an error is called fault latency. The time between the occurrence of an error and the appearance of the resulting failure is called error latency.

D. RELIABILITY

Reliability is the probability of a system performing its intended function under given operating conditions and environments for a specified length of time. It is often expressed by a probability value from 0 to 1 inclusive. Higher value is desirable as it indicates a more reliable system.

The most basic method of achieving reliability is through mature design. Reliability of a device is predicted based on the failure rates when the products are subjected to accelerated conditions. For a complex system, the reliability is determined by each of the individual components. Reliability not only describes the system performance over time, but it includes designing in the ability to maintain, test and support the system throughout its total life cycle. It is accomplished concurrently with other design disciplines by contributing to the selection of the system architecture, materials, processes, and components -- both software and hardware; followed by verifying the selections made by comprehensive analysis and test.

A bathtub curve, as shown in Figure 4, is commonly used to describe the life cycle of a product or device. There are three phases. The decreasing section of the bathtub curve is called the infant mortality phase while the increasing section is the wear-out phase. The frequencies of failure for these two regions are higher than the operating or useful life phase. The useful life phase has a constant failure rate for most devices.

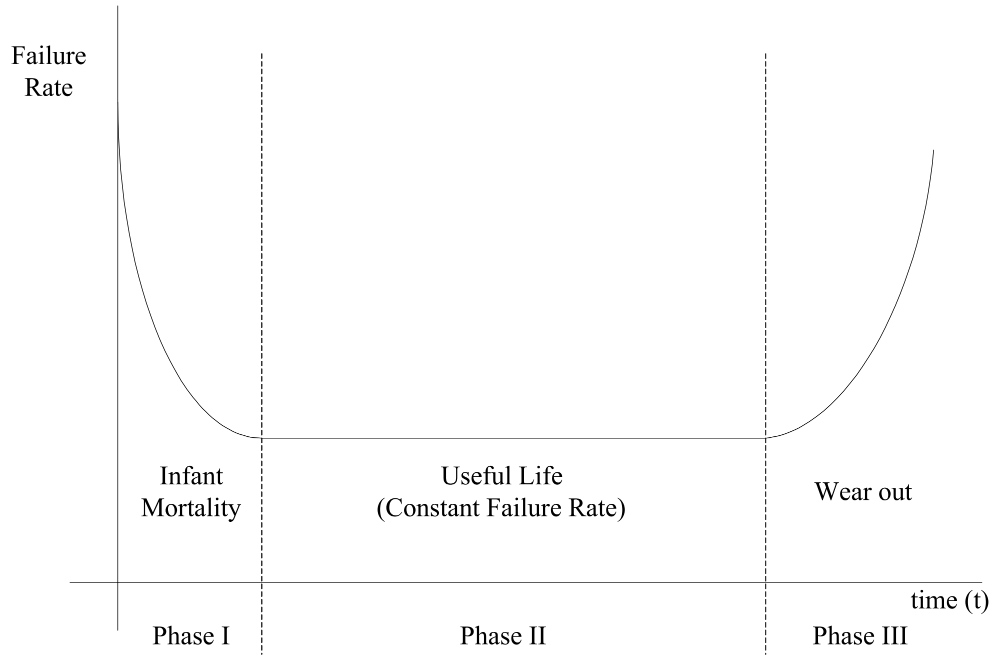

Figure 4. Bath-tub Curve

The reliability of a device is typically computed based on an exponential distribution in the useful life phase as in Equation (1).

$$R(t) = e^{-\lambda t} \tag{1}$$

The parameter λ is the constant failure rate of the system and is defined as the expected number of failures per unit of time. It is the reciprocal of the average system life or Mean Time Between Failure (MTBF) of the system.

E. SECURITY

Security is of little concern for a device until it is used to store or deal with confidential information. It applies mostly to computers, computer peripherals and networking devices. It has become a growing concern, especially in today’s world where there is high connectivity between systems.

Generally, security means to keep any unauthorized personnel from access to systems or data. Computer security is frequently associated with the following five core areas.

- Confidentiality – Ensuring that data is accessible only for reading by authorized persons.
- Integrity – Ensuring that information is not modified or deleted by unauthorized persons without detection by authorized users.
- Authentication – Establishing that users are the persons they claim to be
- Availability – Ensuring that a system is operational and functional at a given moment.
- Non-repudiation – Establishing that neither the originator nor receiver of a message can deny the transmission of the message

Additional terms that are often considered as part of computer security include:

- Access Control – Ensuring that users access those resources and services that they are entitled to access and that qualified users are not denied access to services that they legitimately expect to receive
- Privacy – Ensuring that individuals maintain the rights to control the information collected about them, specifically, how it is used, who uses it, who maintains it, and what purpose it is used for.

F. DIFFERENCE BETWEEN SURVIVABILITY AND OTHER TERMINOLOGIES

In general, the difference between survivability and all other terminologies listed in this thesis is that, survivability includes a well-balanced inheritance of all these quality attributes, from fault tolerance to reliability and from security to availability. It also includes important aspects, which are not lacking in most systems, such as the ability of the system to reconfigure on its own to survive upon attacks or massive failures.

While fault tolerance addresses the issues of accidental fault or combinations of faults through redundancy, it has a limited contribution towards the survivability of the system. Fault tolerance does not ensure survivability of the system. A triple redundant system may have a statistically small probability of failure, assuming independence of faults. However, under a malicious attack aided with the knowledge of the systems internal functions, the assumption of independence is no longer valid. The same attack can be repeated three times quickly to bring down the system. In this case, redundant components must be different so as to improve the susceptibility.

In terms of security, the interest is whether the system is safe from being compromised. Security focuses on building a strong protection on the system, preventing intrusions from taking place, and detecting an intrusion when it occurs. Unlike survivability, security ignores the aspects of recovery and how to ensure the operation of the service during and after an intrusion.

Table 2 shows a comparison of survivability and reliability. Reliability assumes that failures are independent, and as such one component failure will not increase the likelihood of failure of the others. On the other hand, survivability assumes that failures can be correlated. A sufficient number of redundancies will be able to ensure good reliability of the system, but not for survivability. Redundancy only plays a part of ensuring survivability. Unlike survivability, reliability does not include diversity. There is only one defined set of functions. The goal of reliability is to ensure that a failure does not bring down the system, while survivability is to ensure that a failure does not bring down the mission of the system.

	Reliability	**Survivability**
Type of Failure	Independent	Correlated; affecting multiple components
Role of Redundancy	Sufficient with small amount	Not Sufficient (large scale/ impractical)
Recovery Goal	Normal operation (without failure)	Essential services intact; graceful degradation
Diversity	Pre-defined set	Diversified configuration
Security	Not required	Required

Table 2. Comparison of Survivability and Reliability

THIS PAGE INTENTIONALLY LEFT BLANK

III. CONNECTIVITY BASED SURVIVABILITY METRIC

A. GRAPH THEORETIC DEFINITIONS

Since the models defined in this thesis require the use of graphs, this section will cover some basic graph theory definitions. Further definitions can be found in graph theory text such as [ALD00] and [GIB85].

A network is modeled using a graph consisting of nodes representing the communications centers and edges representing links between communications centers. A graph G, which is denoted by (V, E), consists of a set of nodes or vertices V and a set of edges E. Each element of E is an unordered pair (v_i, v_j), where v_i and v_j are elements of V. Each unordered pair indicates two-way communications between two nodes. The unordered pairs create undirected edges. A graph, which consists of undirected edges, is called an undirected graph. An example of an undirected graph is shown in Figure 5, where

$$G = (V, E)$$

$$V = \{a, b, c, d, e, f, g, h\}$$

$$E = \{(a, b), (a, c), (b, c), (b, d), (b, e), (c, e), (d, f), (d, h), (e, f), (e, g), (f, h), (g, h)\}$$

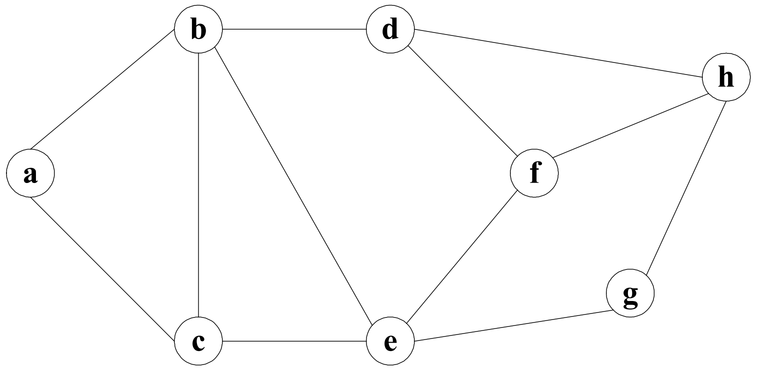

Figure 5. An undirected graph

A loop is an edge that originates and terminates at the same node. Multiple edges, multi-edge or parallel edges occur when two or more edges are incident upon the same pair of nodes. A graph is called a simple graph if it contains no loops or parallel edges;

otherwise it is a general graph. In this work, all graphs are assumed to be undirected simple graphs.

A walk is a sequence of nodes (v_1, v_2, ..., v_n) in which (v_i, v_{i+1}) is an edge of the graph. An example of a walk from node a to node g is (a, b, d, h, g). A trail is a walk with no repeated edges. A path is a walk in which all edges and all vertices on the walk are unique except that the first and last node may be the same, in which case the path is called a cycle. The walk (a, b, d, h, g) is a path. An example of a cycle is (b, d, f, e, d). An a-h path is the path from node a to node h. A hop is equivalent to an edge in a path. In the above example, a path of (a, b, d, h, g) is said to have 4 hops.

Edge-disjoint paths are paths with no edges in common. Node-disjoint paths are paths that share no common nodes other than the source and destination nodes. For instance, (a, b, d, f, e) and (a, c, b, e) are edge disjoint paths while (a, b, d, f, e) and (a, c, e) are node-disjoint paths.

Edges are often labeled with weight values, which represent the physical property of the link. Properties such as cost of an edge (a, b), denoted cost(a, b), or the length of an edge (a, b), denoted length(a, b), are often used when modeling communication networks. However, in this work, unity weights are used for simplicity in analyses.

A cut-set is a set of edges whose removal disconnects the graph. An (a, h)-cut-set is a set of edges whose removal disconnects node a and h. The set of edges {(d, h), (f, h), (g, h), (a, b)} form an (a, h)-cut-set in the above example graph. However, this cut-set is not minimal, since an (a, h)-cut-set can still be formed even without edge (a, b). The set of edges {(d, h), (f, h), (g, h) forms a minimum cut set. A minimum cardinality cut-set is a cut-set that contains the fewest possible number of edges. The edge connectivity of a graph is the size of a minimum cardinality cut-set. The set of edges, {(a, b), (a, c)} forms a minimum cardinality (a, h) cut-set of size two for the above graph displayed in Figure 5.

The degree of a node is the number of edges that are incident from the node. If all nodes in the graph are of the same degree, it is called a regular graph. Two nodes are adjacent if there exists a common edge between them. A complete graph is a graph for which every pair of distinct vertices defines an edge.

B. OPERABILITY

In studying networks, the interest is in the operability of the edge and nodes. However, in this thesis, the coverage will be on the operability of edges only. An edge is operational or available if this link permits communication. The link between the two nodes is said to be in a failed state or simply failed, if communication via this edge is not possible. The probability of an edge to be in its operational state is denoted by p. On the other hand, the probability of an edge failure is $q = 1 - p$.

A path is operational if and only if all edges along the path are operational. Therefore, a path is in a failed state if any of the edges along it has failed. When the paths between source node and the destination node are disconnected, the network is said to be in a failed state.

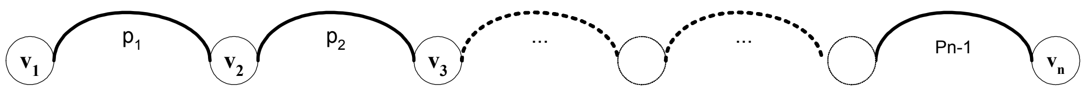

Figure 6. Operability of a path

Hence, when the edge failures are independent, the probability of the path's operability, denoted by P, is

$$P = \prod_{i=1}^{n-1} p_i \tag{2}$$

where n is the number of vertices in the path and p_i represents the probability of operability for the edge from vertex v_i to vertex v_{i+1}.

The probability of a path failure is as in Equation (3)

$$1 - \prod_{i=1}^{n-1} p_i \tag{3}$$

Very often, in networks, there is more than one path from the source node to the destination node. The operability of the connection between the two nodes is dependent on all the paths. In particular, the probability of the connection being operational, denoted by L, is

$$L = 1 - \prod_{j=1}^{m} \left(1 - P_j\right) \tag{4}$$

where m is the number of edge-disjoint paths between the two nodes and P_j is the probability of operability of the j^{th} path.

As long as one path is operational, communication between the two nodes is possible. However, due to the meshed nature of most networks, computation of the operability of the link can be very difficult.

C. A CONNECTIVITY METRIC

The objective of this thesis is to find the most suitable node to host the server agent when a host migration is required. The new host must offer the most optimal locality in the network, such that the connections between the server and its clients are most reliable. It is important to note that different clients of a server have different levels of need for the services offered by the server. Some clients may have a higher priority over the others or some clients may need to access the services more frequent than the others. Therefore, it is important to first measure the suitability of a host on a per-client basis and then synthesize the per-client metrics into a global survivability metric to account for the different levels of client needs.

The interest of this thesis is not on the synthesis process; rather the focus is on quantifying the connection survivability between the server location and a single client. Before proceeding further, it is beneficial to make some assumptions first to simplify the analysis. First, it is assumed that the nodes will not fail. Each node is well protected and shielded against any malicious attack and redundancy is always available to handle system failure due to faults. However, failures can occur to the edges or links, and that may result in the loss of connectivity between the client and the server. Next, edge failures are assumed to have a uniform probability of failure.

The probability that the server node is disconnected from the client is dependent on the probability of having a number of edge failures at the same time. This is formally defined in Equation (5) [XIE02].

$$P_r\{cut(s,d)=1\} = \sum_{i=0}^{\infty} \mathrm{P}_r\{cut(s,d)=1 \mid i \ edge \ failure\} P_r\{i \ edge \ failure\} \tag{5}$$

where s denotes the server node, d denotes the client node and $cut(s,d)$ takes a value of 1 if the client cannot access the service and 0 otherwise.

Equation (5) can be simplified as follows.

$$P_r\{cut(s,d)=1\} = \sum_{i=K_e(s,d)}^{\infty} \mathrm{P}_r\{cut(s,d)=1 \mid i \ edge \ failure\} P_r\{i \ edge \ failure\} \tag{6}$$

where $K_e(s,d)$ is the edge connectivity or the number of edge-disjoint paths between s and d.

Equation (6) states that the number of edge failures must be at least equal to the number of edge-disjoint paths joining the two nodes in order to disconnect the server and the client. This suggests $K_e(s,d)$ may be a good metric for measuring the quantity of the connection between s and d.

Specifically, the survivability of two server locations s_1 and s_2 with respect to the client location d, can be compared in two steps. In the first step, the edge connectivities $K_e(s_1, d)$ and $K_e(s_2, d)$ are compared. The one with larger number of edge-disjoint paths has a higher survivability. When both have the equal K_e values, the comparison proceeds to step 2, in which $P_r\{cut(s_1,d)=1 \mid K_e \ edge \ failure\}$ and $P_r\{cut(s_2,d)=1 \mid K_e \ edge \ failure\}$ are computed and compared. The location with the smaller probability has a higher survivability.

D. COMPUTATION OF K$_E$

The computation of the survivability of the connection between a server and the client is based on the number of edge-disjoint paths between the two nodes, i.e., the K_e value. The greater the number of edge-disjoint paths between the server and client, the better and more reliable the connection is. In an event of multiple edge failures with higher number edge-disjoint paths to the client is less likely to be disconnected from the client. In the reminder, the focus will be on how to compute K_e.

The computation of K_e can be based on a simplified version of the maximum flow algorithm. The maximum flow algorithm determines the capacity constraints or net flow from one node to another in the network. When unit weights are assigned to all the edges in the graph, K_e equals to the maximum flow value.

In general, there are two primary classes of algorithms for computing the maximum flow. They are the augmenting path methods and the preflow-push methods [SKI97]. The augmenting path algorithms repeatedly find a path of positive capacity from the source node to the destination node and add it to the flow. It terminates when there is no more augmenting path. The preflow-push algorithms push flows from one node to another, until the end of the constraints where in-flow must equal to out-flow at each vertex. The preflow-push methods proved to be faster than augmenting path methods. This is due to the fact that multiple paths can be augmented simultaneously in the former case. However, since, preflow-push methods can only be applied to directed graphs, this work is based on one of the augmenting methods, specifically the Ford-Fulkerson maximum flow algorithm.

1. Ford-Fulkerson Maximum Flow Algorithm

Figure 7 outlines the Ford-Fulkerson maximum-flow algorithm finding the maximum flow from s to d. It is the first algorithm invented to solve the maximum flow problem. The complexity of this maximum-flow algorithm is $O(n|E|^2)$, where n is the number of vertices and E is the total number of edges in the graph.

Line 1, 2 and 3 of the algorithm nullifies the flow and initializes the graph, which requires $O(|E|)$-time. Line 4 repeats Line 5 to Line 9 as long as an augmenting path p is found in the graph between the server and client node. If an augmenting path p is found in Line 5, then the capacity of p, $c_f(p)$, will be computed in Line 6 based on the minimum value of the weight of the edge, $c(u,v)$ and the flow of the weight, $f(u,v)$. The flow will be updated according to the mathematical definitions in Line 10. This will continue until there are no more augmenting paths found in the graph. The output of this algorithm is the summation of flow capacities of all the paths.

```
    procedure MaximumFlow(s, d)
    begin
        // Part I:  Setup
1.      start with null flow: $f(u,v) \leftarrow 0 \; \forall (u,v) \in E$;
2.      initialize $c(u,v) \leftarrow edge\ capacity\ (weight)\ of\ edge(u,v) \in E$;
3.      initialize residual graph to be original graph;

        // Part II: Loop
4.      repeat
5.          search for augmenting path from s to d in residual graph;
6.          if (path $p$ found) then
7.              $c_f(p) \leftarrow \min\{c(u,v) \mid (u,v) \in p\}$;
8.              for (each $(u,v) \in p$) do
9.                  $f(u,v) \leftarrow f(u,v) + c_f(p)$
10.                 remove $(u,v)$ from residual graph;
11.     until (no augmenting path);
12.     initialize $flow = 0$;
13.     for (each $(u,v) \in E$) do
14.         $flow = flow + f(u,v)$;
15.     return $flow$;
    end
```

Figure 7. The Maximum-Flow pseudo code

2. Breadth-First Search

Figure 8 shows breadth-first search. Breadth-first search is used in the maximum flow algorithm to find augmenting paths because it ensures the paths chosen are of the minimum length. The complexity of this algorithm is $O(n+|E|)$.

```
    procedure BreadthFirstSearch(s, d)
    begin
1.      initialize container $L_0$ to contain vertex $s$;
2.      initialize $e \leftarrow unexplored$; for all $e \in E$;
3.      $i \leftarrow 0$;
4.      while $L_i$ is not empty do
5.          create container $L_{i+1}$ to initially be empty;
6.          for each vertex $v$ in $L_i$ do
7.              for each edge $e$ incident on $v$ do
8.                  if edge $e$ is unexplored then
9.                      let $w$ be the other endpoint of $e$;
10.                     if vertex $w$ is unexplored then
11.                         label $e$ as a discovered edge;
12.                         insert $w$ into $L_{i+1}$;
13.                     else
14.                         label $e$ as a cross edge;
15.         $i \leftarrow i+1$;
16.     initialize child node to destination node, $c \leftarrow d$;
17.     initialize a list, $List$ to contain $d$;
18.     while $c \neq s$ do
19.         for each edge $e$ ending on $c$ do
20.             if edge $e$ is discovered then
21.                 $List \leftarrow$ the other end vertex of $e$;
22.     return List;
    end
```

Figure 8. Breadth-First Search pseudo code

The breadth-first search algorithm starts by defining the starting vertex s as an ‘anchor’. In the first round, it will search all the edges that are only one edge away from the anchor. Subsequent rounds will increase the search radius by one edge each, till the client node (d) is found. Tracing back the nodes through the explored edges give the sequence of the vertex for the path. With this approach, it ensures that the path found is of the minimum length.

E. MINIMUM CUT (MIN-CUT)

Recall that K_e by itself cannot provide the desired level of granularity in comparing and ranking the suitability of candidate server nodes because the K_e value of more than one node can be the same. In that case, the comparison has to go further, based on this conditional probability $P_r\{cut(s,d)=1 \mid K_e\ edge\ failure\}$. To compute that probability, it is necessary to find the number of min-cut between the server node and the client. Clearly, the higher the number of min-cuts, the larger the value of $P_r\{cut(s,d)=1 \mid K_e\ edge\ failure\}$ is.

Observe that a smaller than K_e number of edge failures will not disconnect the server node and the client. This is because it requires the removal of at least one critical edge in each of the edge-disjoint paths in order to disconnect the two nodes. Critical edges are edges that do not have alternate edges or paths. For instance, Figure 9 shows two edge-disjoint paths from node a to node f: (a, b, f) and (a, c, d, f). For the second path, (a, c) is a critical edge while, (c, d) and (d, e) are not. There exists an alternate path from node c to node f.

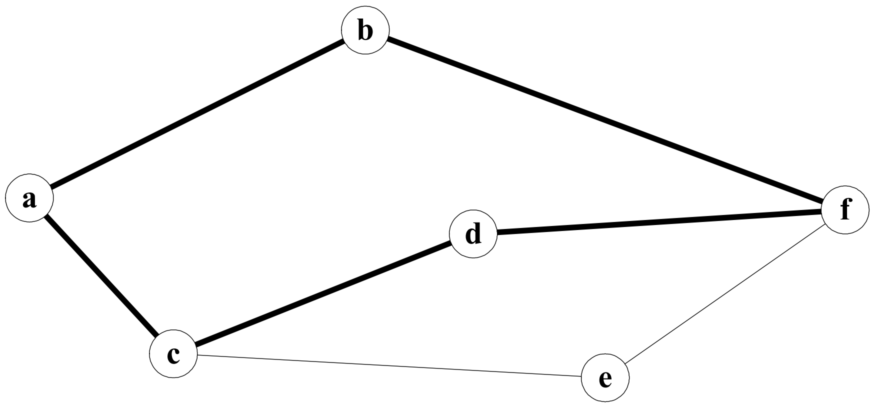

Figure 9. Example of critical edges

Therefore, the number of critical edges determines the survivability of the connection between the server node and the client. The more critical edges exist in a path, the survivability is lower for that path, as it results in a higher number of min-cuts.

Figure 10 shows a client at node f, and the choice of the server node is to be determined. The highest achievable K_e is 3, since the constraint is on the degree of the client node, node f, in this case. The results based on K_e and Min-cuts computation are tabulated in Table 3. Node a, c, d and e are potential server hosts for the client; each of them gave a K_e value of three for the client node f. However, checking the number of min-cuts for each potential node, node c is the best choice for hosting the server, since it has the minimum number of min-cuts among the nodes with the highest K_e.

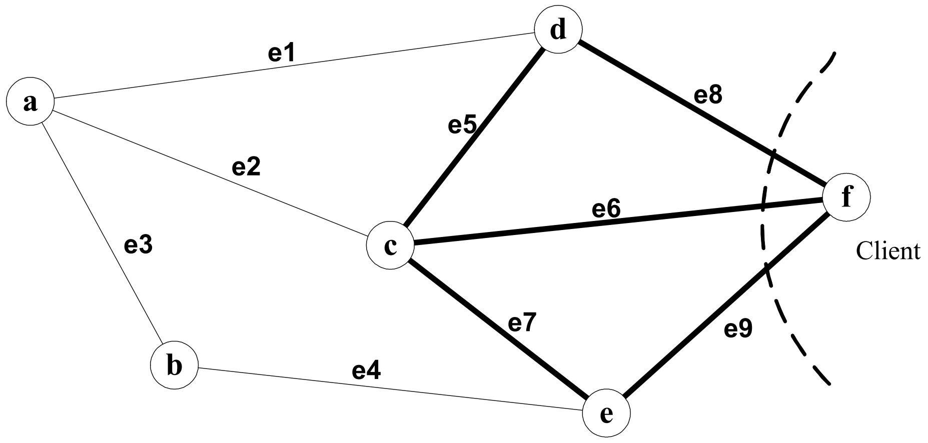

Figure 10. Minimum Cut-sets

Potential Server Node	Edge Connectivity, K_e	Min-cuts
a	3	(e1, e2, e3), (e1, e,2, e4), (e6, e8, e9)
b	2	(e3, e4)
c	3	(e6, e8, e9)
d	3	(e1, e5, e8), (e6, e8, e9)
e	3	(e3, e7, e9), (e4, e7, e9), (e6, e8, e9)

Table 3. Min-cuts for each potential server node

1. Algorithm Used to Find the Min-cuts

At this point in time, there is still no known algorithm of polynomial complexity to find the min-cuts. In Chapter IV, several heuristic models are developed to rank nodes with the same K_e value without actually computing the number of min-cuts. To assist the testing of the heuristic models, a brute-force algorithm was implemented by another graduate student, Baris Aktop, to enumerate the min-cuts between the two nodes. Basically, this method regenerates a graph by removing K_e edges from the original one, and runs the maximum flow algorithm to check if the K_e value reduces. If it does, a min-cut is defined. This is repeated for every possible combination of K_e-edges. The complexity is $O(E^{K_e})$. The pseudo code is shown in Figure 11.

```
     procedure FindMinCuts()
     begin
1.       $count = 0$;
2.       $numOfCutSets = 0$;
3.       while $count \,!= {}^{E}C_{K_e}$ do
4.           select $K_e$ edges;
5.           $G' \leftarrow remove\ selected\ K_e\ edges\ from\ G$ ;
6.           maximum flow algorithm to find $K_e'$;
7.           if $K_e' = K_e$ then
8.               $numOfCutSets \leftarrow numOfCutSets + 1$;
     End
```

Figure 11. Pseudo code for finding the Min-cuts

THIS PAGE INTENTIONALLY LEFT BLANK

IV. HEURISTIC MODELS FOR RATING QUALITY OF CONNECTIVITY

A. PURPOSES OF HEURISTIC MODELS

The Min-Cut approach is one way of further rating the survivability of the nodes that have the same Ke value. However, it is shown that the enumerating of min-cuts between two nodes is NP-hard. Hence, it is of the interest of this thesis to find a heuristic way of performing this rating without the enumeration process.

A few heuristic models are defined in this chapter. Each of these models is being compared against one another to determine which heuristic model gives the most accurate estimation of the survivability of the hosting node.

B. DEGREE MODEL

The first heuristic model proposed is called the degree model. It is the simplest model with a complexity of $O(n)$, where n is the number of vertices in the graph. Basically, this model assumes that a node having a higher degree gives the node a higher survivability value. With a greater number of outgoing edges, the node can reach more clients. Furthermore, more edges means it is less likely for the node to be disconnected from the others.

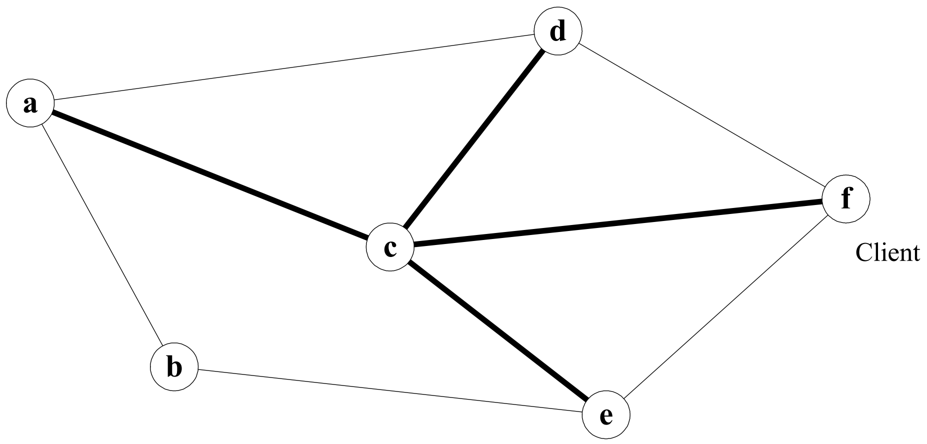

Figure 12. Selection of Host based on Degree of the nodes

Figure 12 shows the same graph as the one used in the previous chapter for computing the number of min-cuts. With the same client at node f, the degree heuristic model determines that the SAAM server should be placed at node c, since it has a degree of four, which is the largest degree among all the nodes with the highest K_e value.

C. HOP COUNT MODEL

The hop count model basically counts the number of hops in all the edge-disjoint paths. The node, which requires the least total number of hops to reach the client, is deemed the best choice to host the server. The hop count model is also known as the proximity model, because a lower hop count usually means the node is closer to the client.

The hop count model is based on the fact that the smaller number of edges a path has, the lower the chance that random edge failures will disconnect the path.

The same example is reproduced in Figure 13. With the same client node f, the server node is determined with the hop count model. Both nodes b and c yield the lowest number of hop count of five. However, node b has a K_e value of two, whereas, node c has a K_e value of three. Since the K_e value is the primary criterion and no other node has a K_e value greater than three, node c is the best choice.

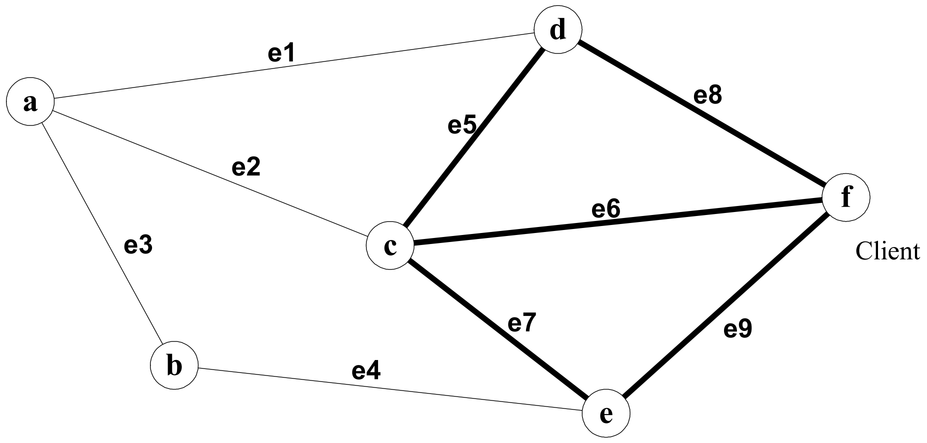

Figure 13. Selection of Host based on Hop Count

Potential Server Node	Edge Connectivity, K_e	Path set	Total Hop Count
a	3	(e1, e8), (e2, e6), (e3, e4, e9)	7
b	2	(e4, e9), (e3, e1, e8)	5
c	3	(e6), (e5, e8), (e7, e9)	5
d	3	(e8), (e5, e6), (e1, e3, e4, e9)	7
e	3	(e9), (e7, e6), (e4, e3, e1, e8)	7

Table 4. Hop Count Results

D. P_E MODEL

The P_e model determines the survivability of the connection based on the number of critical edges in each edge-disjoint path between the two nodes. An edge is critical if its failure will disconnect the path. So, the less critical edges, the lower the probability of path disconnection.

In the P_e model, the goal is to approximate $P_r\{cut(s,d)=1 \mid K_e \ edge \ failure\}$, the probability of connection failure given K_e number of edge failures. The connection between the two nodes fails if and only if the edge failures disconnect all the edge-disjoint paths. In other words, each of the edge failures must be a critical edge of a different edge-disjoint path for the connection failure.

Consider K_e edge-disjoint path and Ke edge failures. The probability of the first edge failure to be at a critical edge of the first path is $\frac{number \ of \ critical \ edges \ in \ first \ path}{total \ number \ of \ edges}$, the probability of the second edge failure to be at a critical edge of the second path is $\frac{number \ of \ critical \ edges \ in \sec ond \ path}{total \ number \ of \ edges - 1}$, and so on, through the K_e path. The order of the paths cut can be selected in K_e factorial ways. Hence, if the number of critical edges of each edge-disjoint path is independent of critical edge failures on the other paths, $P_r\{cut(s,d)=1 \mid K_e \ edge \ failures\}$ can be derived by Equation (7).

$$P_e = K_e! \times \prod_{i=1}^{K_e} \left(\frac{C_{p_i}}{E - i + 1}\right) \tag{7}$$

where P_i is the i^{th} edge-disjoint path, C_{P_i} is the number of critical edges in the i^{th} path, and E is the total number of edges in the network. In general, the above assumption that the number of critical edges for a path remains constant in the presence of critical edge failures on the other paths is not true. That is why P_e is a heuristic.

Figure 14 illustrates the computation of the P_e value using Equation (7). With respect to the client node f, the P_e value of node c is derived as follows. The K_e value is 3, and the three paths are:

P_1 = (e6)

P_2 = (e5, e8)

P_3 = (e7, e9)

In Path 1, the only edge e1 is the critical edge. In Path 2, edge e8 is the critical edge but not e5, since it has an alternate path (e2, e1). Likewise, edge e9 is the critical edge of Path 3. Hence, P_e value is computed as follows:

$$\begin{aligned} P_e &= 3! \times \left(\frac{1}{9} \times \frac{1}{8} \times \frac{1}{7}\right) \\ &= 0.011905 \end{aligned}$$

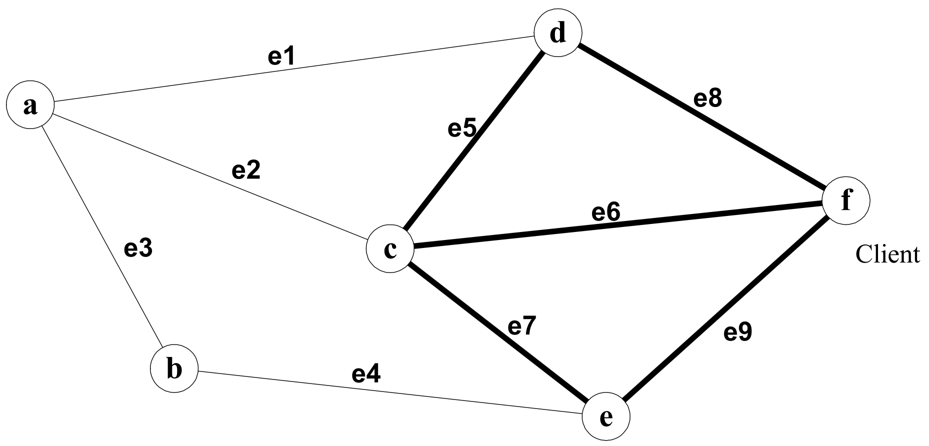

Figure 14. Selection of Host based on P_e

The P_e values for the rest of the nodes are computed and included in Table 5 below. The results show that node c is once again the best server node for client node f.

Potential Server Node	**Edge Connectivity, K_e**	**Critical Edges of each path**	**P_e**
a	3	(**e1**, **e8**), (**e2**, **e6**), (**e3**, **e4**, **e9**)	0.142857
b	2	(**e4**, e9), (**e3**, e1, e8)	0.027778
c	3	(**e6**), (e5, **e8**), (e7, **e9**)	0.011905
d	3	(**e8**), (**e5**, **e6**), (**e1**, e3, e4, **e9**)	0.047619
e	3	(**e9**), (**e7**, **e6**), (**e4**, **e3**, e1, **e8**)	0.071429

Table 5. P_e results

1. Algorithm Used to Compute P_e

The algorithm used to compute the P_e value is shown in Figure 15. It takes the augmenting paths, i.e., shortest edge-disjoint paths when unit weights are used, found by the breadth-first search in the computation of K_e, and finds the number of critical edges for each of the paths. Finding the number of critical edges is the heart of the P_e model.

The algorithm starts by creating a graph G' as described in Line 1. Graph G' is a duplicate of the original graph G, less the edges used in constructing the augmenting paths. For each path, Line 3 initializes the number of critical edges, C_{p_i} to the length of the path. Then, at Line 4, each edge and consecutive sequence of edges, $link(u \rightarrow v)$ are checked for an alternate path in graph G'. If an alternate path exists, C_{p_i} is reduced by a value equal to the distance between vertex u and v; this corresponds to Line 7 in the pseudo code. To improve the efficiency, the longest sub-path is explored first, and if there is an alternate path for that sub-path, then all the edges in the sub-path are not critical. With the critical edges for each path known, P_e can be easily computed using Equation (7).

```
    procedure FindPe(s, d)
    begin
1.      $G' \leftarrow remove\ edges\ of\ augmenting\ path\ from\ G$ ;
2.      for each $P_i \in P_{K_e}$ do
3.          $C_{p_i} \leftarrow HopCount(P_i)$ ;
4.          initialize $u \leftarrow s$ and $v \leftarrow d$ ;
5.          while $u \neq d$ do
6.              use Maximum flow algorithm to find an alternate path from
                $u$ to $v$ in $G'$ ;
7.              if alternate path exists then
8.                  $C_{p_i} \leftarrow C_{p_i} - HopCount(u, v)$ ;
9.                  $u \leftarrow v$ ;
10.                 $v \leftarrow d$ ;
11.             else
12.                 $v \leftarrow previous\ node\ of\ d\ on\ P_i$ ;
13.                 if $v = u$ then
14.                     $u \leftarrow next\ node\ on\ path\ P_i$ ;
15.                     $v \leftarrow d$ ;
16.     compute $P_e = K_e! \times \prod_{i=1}^{K_e} (\frac{C_{p_i}}{E - i + 1})$ ;
    end
```

Figure 15. Pseudo code for P_e computation

E. 'RESISTANCE' RULE MODEL

The resistance rule models the network as a series of resistors connected in a meshed circuit. As shown below, Figure 16 can be represented by the circuit diagram as shown in Figure 17. Each edge corresponds to a unit resistor. The resistance of the path can be calculated based on the source node, destination node, and the path taken. The resistance of path (a, f, g, h) is 3 units. Intuitively, a lower resistance between the source node and the destination node is desirable as it signifies a lower possibility of an edge failure that will disconnect the source and target nodes.

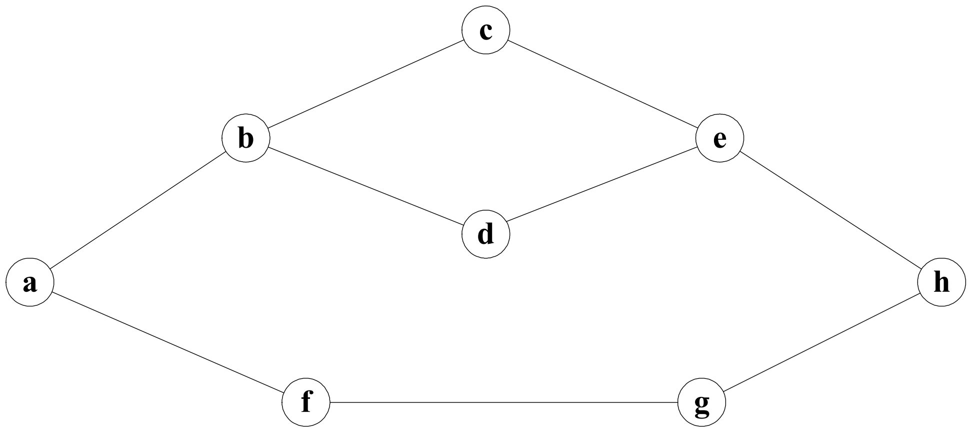

Figure 16. A graph

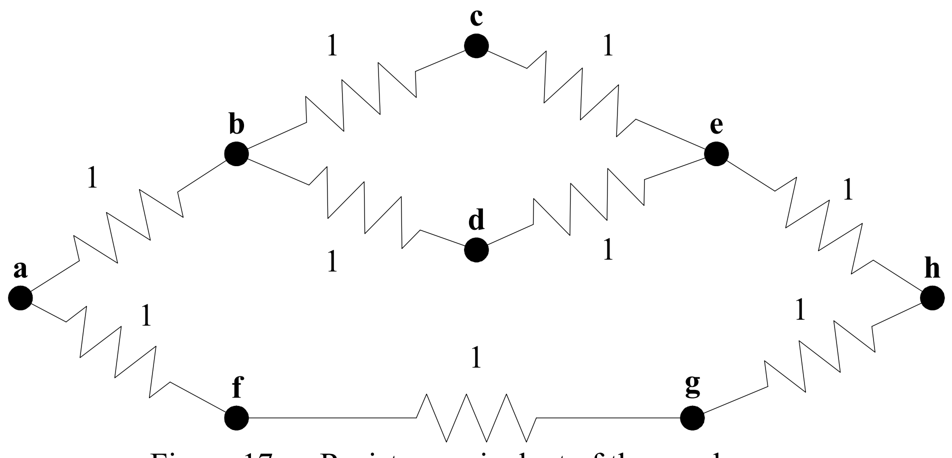

Figure 17. Resistor equivalent of the graph

However, the interest is not on a single path, rather, the overall connectivity between the source and the destination nodes. Hence, alternate paths need to be considered as well. The possibility of failure or the resistance between the source and destination nodes can be computed by applying the parallel resistance rule as in Equation (8),

$$R_{equivalent} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \dots\dots + \frac{1}{R_n}} \tag{8}$$

where R_1, R_2, …, R_n are parallel paths, which have a common source and destination nodes.

Using parallel resistance rule, the resistance from node b to e is calculated to be one unit. This gives path (a, b, e, h) a resistance of three units as well. Thus, with two parallel paths of three units, the overall resistance from node a to node h is computed as 1.5 units. From this, it can be seen that the resistance rule gives a good estimation on the possibility of failure between the two nodes of interest.

However, the resistance rule does not identify the critical edges of each path. Furthermore, application of resistance rule can be very complicated in highly meshed network, like the one shown in Figure 18. Therefore, the model is not considered further.

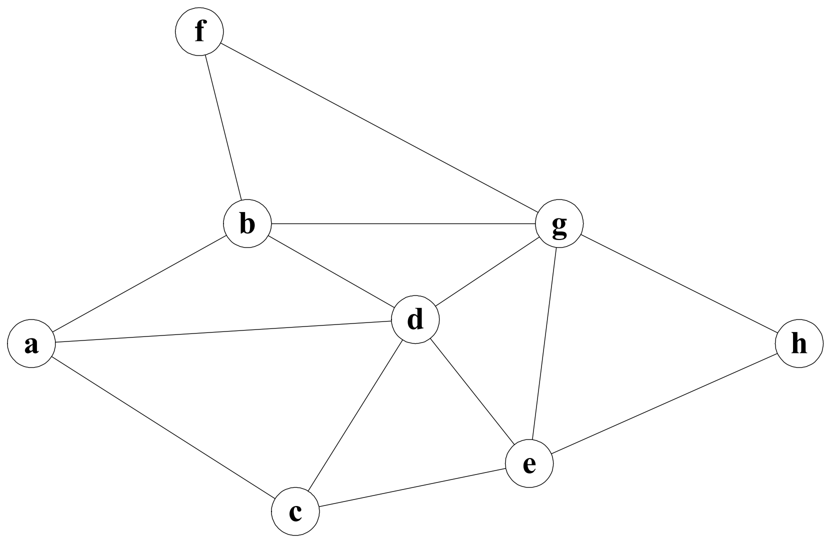

Figure 18. Typical network topology

V. EXPERIMENTS AND ANALYSES

A. SIMULATION PROGRAM

In order to carry out the experiments to determine the validity of each of the heuristic models, a software simulation program is required. This software program needs to have the capability of generating random graphs in addition to user-inputted graphs. To save the effort, an open source program, called the Java-based Graph Algorithm Platform (JGAP) was downloaded from the Internet at http://im.ncnu.edu.tw/~tsai/definite/JGAP/JGAP.html.

JGAP has most of the basic needs required by this thesis. It allows the user to create both directed and undirected graphs, through its user-friendly Graphical User Interface. Random graphs can be generated as well; the user just needs to input the number of vertices of the graphs and the density for the edges. A higher density value will generate a random graph with more edges. Other than that, there are already a number of graph algorithms built into the program. They include well-known algorithms like, Breadth First Search, Depth First Search, Ford-Fulkerson Maximum flow algorithms. With these algorithms in place, much development time is saved.

After some comprehension and exploration of the source code, modifications to the original program were carried out to meet the thesis needs. First, the P_e algorithm is incorporated; this helps to verify the various results that were computed manually, as well as to verify the correctness of the algorithm. The degree and hop count heuristic algorithm were then included. The random creation of graphs was modified so that the program can now generate random graphs with a good number of edge-disjoint paths between two vertices. This is important because too many edges originating or ending at one node does not resemble an actual network topology, while too few edges will result in disconnected nodes.

With the capability of generating more realistic random graphs, the program was enhanced to allow multiple simulation runs, allowing the user to run continuously for a specified number of graphs. The whole process of generating new graphs and computing the results for each heuristic model was thus automated. The results were then stored in an external file for analysis purposes. More details of the experiment results will be covered later in this chapter.

Subsequently, data analysis code was included into the program as well. This saves the user lots of time and effort as compared to analyzing the results manually. Furthermore, the amount of results to be analyzed can be large, when the number of graphs is large.

Figure 19 shows a screen capture of the simulation program used for this thesis.

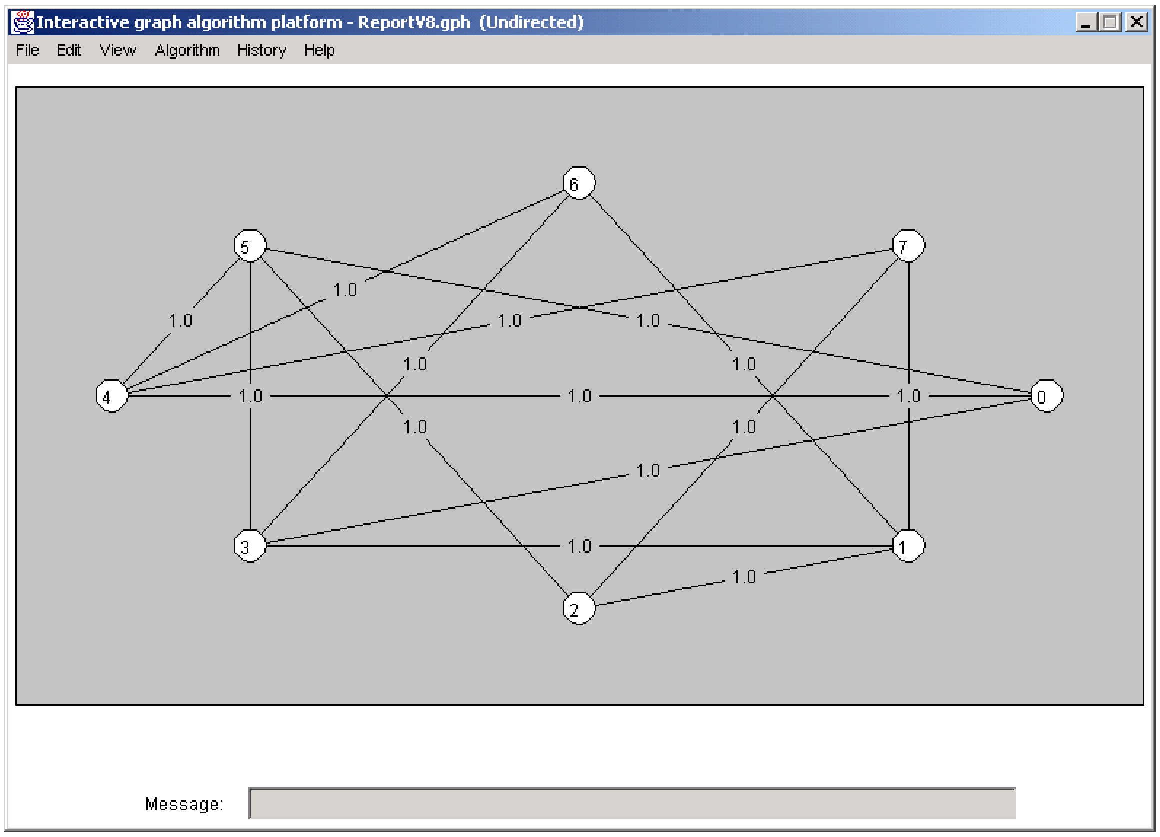
Interactive graph algorithm platform - ReportV8.gph (Undirected)
File
Edit
View
Algorithm
History
Help
0
1
2
3
4
5
6
7
1.0
Message:

Figure 19. Simulation Program

B. DESCRIPTION OF SIMULATION RESULTS

The results from the simulation runs were saved in external files. By doing so, it allowed the separation of the simulation runs, usually time consuming, from the data analysis processes. It also allowed new analysis processes to be applied off line, without the need to re-run the simulation again.

Figure 20 shows an example of the results stored in an external file. Each block of results starts with a header naming the fields in the subsequent rows of data, which are separated by commas and semicolons. The header begins with the start node, which represents the potential server node, while the second number represents the end node or the client node. It can be observed that each block of results is defined based on the client node, since the end node is common in the same block.

The fourth column, after the semicolon, tells the K_e value of the start node to the end node. This value is important because it is the primary factor to be considered when ranking the potential server node.

The fifth column denotes the number of min-cuts for the connections between the start and the end node. A lower value is always more desirable, because it means that there is a lower value for $P_r\{cut(s,d)=1 \mid K_e \ edge \ failures\}$.

The sixth column indicates the degree of the start node or the server node.

The seventh column shows the total number of hops for all the edge-disjoint paths between the start node and the client node.

The last column or the eighth column specifies the P_e value from the start node to the end node. More details on the analysis of the results will be discussed in the next section.

```
! startNode,endNode,TotalEdges;ke,Cutset,degree,HopCount,Pe
1,0,14;4,2,4,  8,0.00799200799200799
2,0,14;3,1,3,  7,0.002747252747252747
3,0,14;4,2,4,10,0.011988011988011988
4,0,14;4,2,4,10,0.01598401598401598
5,0,14;3,1,3,  7,0.002747252747252747
6,0,14;3,1,3,  8,0.002747252747252747
7,0,14;3,1,3,  6,0.002747252747252747

! startNode,endNode,TotalEdges;ke,Cutset,degree,HopCount,Pe
0,1,14;4,2,4,  8,0.00799200799200799
2,1,14;3,1,3,  7,0.002747252747252747
3,1,14;4,2,4,  9,0.00799200799200799
4,1,14;4,2,4,10,0.03596403596403597
5,1,14;3,1,3,  7,0.002747252747252747
6,1,14;3,1,3,  7,0.002747252747252747
7,1,14;3,1,3,  6,0.002747252747252747

! startNode,endNode,TotalEdges;ke,Cutset,degree,HopCount,Pe
0,2,14;3,1,4,7,0.002747252747252747
1,2,14;3,1,4,7,0.002747252747252747
3,2,14;3,1,4,7,0.002747252747252747
4,2,14;3,1,4,7,0.002747252747252747
5,2,14;3,2,3,7,0.010989010989010988
6,2,14;3,2,3,7,0.010989010989010988
7,2,14;3,2,3,8,0.021978021978021976

! startNode,endNode,TotalEdges;ke,Cutset,degree,HopCount,Pe
0,3,14;4,2,4,10,0.011988011988011988
1,3,14;4,2,4,  9,0.00799200799200799
2,3,14;3,1,3,  7,0.002747252747252747
4,3,14;4,2,4,  9,0.00799200799200799
5,3,14;3,1,3,  7,0.002747252747252747
6,3,14;3,1,3,  6,0.002747252747252747
7,3,14;3,1,3,  6,0.002747252747252747

…
…
```

Figure 20.   Results of Simulation runs

C. ANALYSIS OF RESULTS

With the results collected from the simulation runs, various analysis methods were applied to them. However, the results did not allow direct comparison, since they are not of the same value type. So, the results were sorted and ranked into a number of groups as shown in the example below.

With reference to Figure 20, the first block of data shows the results of the possible servers for a client at node 0. Looking at K_e column, the best choice to host the server, other than the client node itself, will be either node 1, 3, or 4, since they have the highest K_e value.

The degree model ranks nodes 1, 3 and 4 first, and nodes 2, 5, 6, and 7 last. Using the hop count model, node 1 is ranked first, followed by nodes 3 and 4. The P_e model ranks node 1 first because of its lowest P_e value among all, followed by node 3, and then node 4. The ranking of each factor is summarized in Table 6.

Ranking Group	Min-Cut	Degree	Hop Counts	P_e
First	1, 3, 4	1, 3, 4	1	1
Second	2, 5, 6, 7	2, 5, 6, 7	3, 4	3
Third	-	-	7	4
Fourth	-	-	2, 5	2, 5, 6, 7
Fifth	-	-	8	-

Table 6. Ranking of Results

With the ranking in place for each model, ranking analyses can be applied to each of heuristics using the min-cut results as a reference. In this work, two such analyses were carried out to determine the accuracy of each heuristic model. They are the Best Nodes Analysis and the Ranking Analysis.

1. Best Nodes Analysis

This analysis compares the nodes in the first rank group of each model with that of the Min-cut's. If a node is ranked in the first group in a model, it must also be ranked in the first group of the Min-cut's ranking. Otherwise, it will be considered as an error. For instance, if the Min-cut reference ranks node 1, 3, and 4 in the first group and a heuristic model ranks nodes 1, 2, 3 and 5 in the first group, then the number of errors will be two. This is because comparing with the Min-cut reference ranking, nodes 2 and 5 are incorrectly ranked by the heuristic model, since they are not ranked first by the Min-cut ranking.

By counting the number of first-ranking errors made by each model, the accuracy of each model can be determined. A low error value means the heuristic model is accurate in determining the server-hosting node.

```
    function BestNodesAnalysis( $H[], M[]$ )
    begin
1.      $noOfErrors = 0$;
2.      for each node $n_H$ in first RankGroup of $H[]$ do
3.          $nodeFound \leftarrow false$;
4.          for each node $n_M$ in first RankGroup of $M[]$ do
5.              if $n_H = n_M$ then
6.                  $nodeFound \leftarrow true$;
7.          if $nodeFound = false$ then
8.              $noOfErrors \leftarrow noOfErrors + 1$;
9.      return $noOfErrors$;
    end
```

Figure 21. Best Nodes Analysis pseudo code

Figure 21 shows the pseudo code for Best Nodes Analysis. The algorithm takes in the heuristic model's node-ranking set $H[]$ as well as the Min-cut reference's node-ranking set $M[]$. Line 1 initializes the error counter $noOfErrors$. Line 2 checks the nodes ranked first in $H[]$ against the nodes ranked first in $M[]$. The variable $nodeFound$ is initialized to false before the check commences as described in Line 4 and 5. Line 6 sets the $nodeFound$ to true if the same node is ranked first in $M[]$ as well. If not, Line 7 will take that as an error and increase $noOfErrors$ by one. This is repeated for every node ranked first in $H[]$. Before returning to the calling routine, Line 7 passes back the number of errors found to the calling routine.

2. Ranking Analysis

Unlike Best Nodes analysis, the Ranking Analysis is more thorough. The ranking analysis not only looks into the first-ranking group of each model, it goes through the entire ranking set of the heuristic models, comparing each node's rank with that of the Min-cut reference set.

In this analysis, the focus is to find the frequency of error occurrence as well as the average magnitude of a ranking error. Like the Best Nodes Analysis, the frequency of error occurrence counts the number of nodes ranked incorrectly, except checking is carried out on each of the nodes instead of just the ones in the first rank group. The magnitude of the error is computed based on the best ranking position as defined in Equation (9).

$$errorMagnitude(n) = BestPosition(M, n) - BestPosition(H, n) \quad (9)$$

where n is the node in error, M is the Min-cut ranking set, H is the heuristic model ranking set. $BestPosition(x, y)$ corresponds to the best ranking position node y can be in set x, and this depends on the number of nodes ranked ahead of y in x. After the error magnitude is tallied, n will then be removed from both ranking sets, so that the error will not affect the analysis of the subsequent nodes in the heuristic rankings.

For instance, Table 7 shows the ranking groups of the Min-cut and the heuristic model. There is a single error occurrence for node 6. $BestPosition(M,6) = 6$ and $BestPosition(H,6) = 1$, so the magnitude of the error for node 6 is five.

Notice that after removing node 6 from the Min-cut and heuristic model rankings eliminates the error induced by node 6 and as are result both rankings are the same. The analysis of nodes 2, 5 and 7 can then be correctly carried out.

Ranking Group	**Min-cut**	**Heuristic Model**
First	1, 3, 4	1, 3, 4, 6
Second	2, 5	2, 5
Third	6, 7	7

Table 7. Illustration example for Ranking Analysis

Figure 22 shows the Ranking Analysis pseudo code. It takes in both the heuristic model's ranking set $H[]$ and the Min-cut's ranking set $M[]$ as inputs. Unlike the Best Node Analysis, this analysis computes the magnitude of the errors found.

The algorithm starts by initializing the variable, *errorMagnitude*, as described in Line 1. Each node in $H[]$ is checked, starting from first ranking group. The best possible position of the node is computed in Line 3 based on the number of nodes ranked ahead of it. A function *ComputeBestPosition* is written to carry out this task and will be described in detail later. The same node is then searched in the Min-cut reference model as shown in Line 4 and 5. When the node is found in Min-cut's ranking, Line 6 will compute its best possible position in the Min-cut reference set. If the heuristic model gives the node a better position than the Min-cut, the difference in the positions is calculated as described by Line 8; after that, the error node is removed from both the heuristic model and the Min-cut model ranking sets. This is to ensure that the error caused by one node will not propagate to the others. Once, all the nodes in the heuristic model have been analyzed, Line 11 returns the total error magnitude to the calling program.

```
    function $RankOrderAnalysis(H[], M[])$
    begin
1.      $errorMagnitude = 0$;
2.      for each node $n_H$ in $H[]$ do
3.          $R(n_H) \leftarrow ComputeBestPosition(H[], n_H)$;
4.          for each node $n_M$ in $M[]$ do
5.              if $n_H = n_M$ then
6.                  $R(n_M) \leftarrow ComputeBestPosition(M[], n_M)$;
7.                  if $R(n_H) < R(n_M)$ then
8.                      $errorMagnitude \leftarrow errorMagnitude + \left(R(n_M) - R(n_H)\right)$;
9.                      remove node $n_H$ from $H[]$;
10.                     remove node $n_M$ from $M[]$;
11.     return $errorMagnitude$;
    end
```

Figure 22. Ranking Analysis pseudo code

In Figure 23, the *ComputeBestPosition* function computes the best possible position of a node based on the number of nodes that are ranked ahead of it. This function takes in a ranking set and a node number.

The function first initializes the counter *noOfNodeAhead* in Line 1. It then searches for the occurrence of the node in the array of rankings in Line 2 and 3. When the node is found, Line 5 will compute its best possible position by counting the number of nodes of higher rank and adding one. Line 6 then returns the position to the calling program. When the node is not found, Line 7 increases the nodes counter *noOfNodeAhead* by the size of the rank group the function has just searched.

```
    function $ComputeBestPosition(V[], nodeNo)$
    begin
1.      $noOfNodeAhead = 0$;
2.      for each rankGroup $v$ in $V[]$ do
3.          for each node $x$ in rankGroup $v$ do
4.              if $x = nodeNo$ then
5.                  $actualRank \leftarrow noOfNodeAhead + 1$;
6.                  return $actualRank$;
7.          $noOfNodeAhead \leftarrow noOfNodeAhead + sizeOf(v)$;
    end
```
Figure 23. Pseudo Code to Compute Best Position

D. SIMULATION AND ITS RESULTS

1. Simulation

Simulations were run by randomly creating graphs of various numbers of vertices, applying heuristic models to the graphs, and analyzing the results generated by each heuristic model. In this thesis, a hundred random graphs were generated for each of the node counts: 7, 8, 9 10 and 12. Larger graphs could be generated and analyzed as well, however, the time required to obtain the Min-cut ranking sets would take too long because of the exponential complexity of the algorithm. Hence, they are not included in this thesis.

2. Results

Table 8 shows the results obtained after applying the Best Nodes Analysis, while Figure 24 displays the graphical representation of the results. The values are percentages of client locations with first-ranking errors. Recall that ranking is done on a per client location basis. For example in the case of 7 vertices, there are total of 700 possible client lications, 5.18% of which has a first ranking error. The results indicate that for all network sizes, the P_e model produces the best result, with almost 0% errors.

Number of Vertices	Best Nodes Analysis (Percentage of ranking sets with first-ranking errors)		
	Degree Model	Hop Count Model	P_e Model
7	5.18	11.13	0
8	6.55	10.67	0
9	7.01	8.71	0.07
10	7.51	12.13	0
12	7.61	8.70	0

Table 8. Results based on Best Nodes Analysis

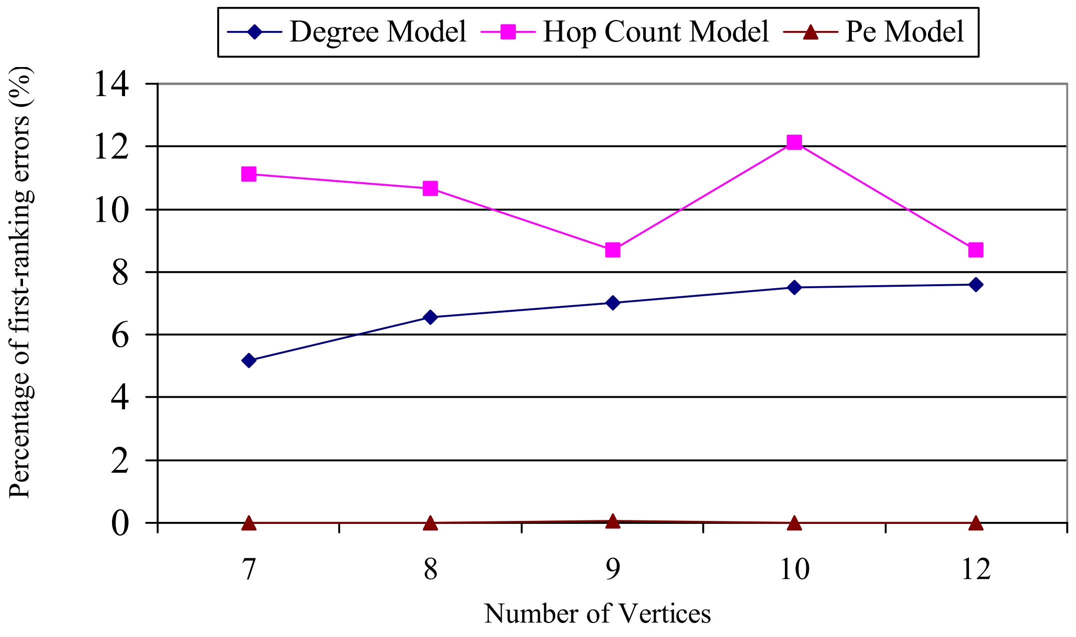

Figure 24. Best nodes Analysis Graphical results

The results obtained from the Ranking Analysis are tabulated into Table 9 and 10. Table 9 tabulates the percentage of client locations with ranking order errors, while Figure 25 graphically displays the frequency of errors. Comparing the analysis results of the different heuristic models, it can be concluded that the hop count model generates the most errors. More than a third of the ranking sets are incorrect. In contrast, the P_e model produces the best estimation of the ranking order, compared to the Min-cut reference, with more than 92% of the sets correctly ranked.

Number of Vertices	Ranking Analysis (Percentage of client locations with ranking order error)		
	Degree Model	Hop Count Model	P_e Model
7	17.43	35	0.57
8	21.5	38.13	2.63
9	23	40.56	5.33
10	27.4	48.6	6
12	26.83	46.67	7.92

Table 9. Results based on Ranking Analysis (Frequency of errors)

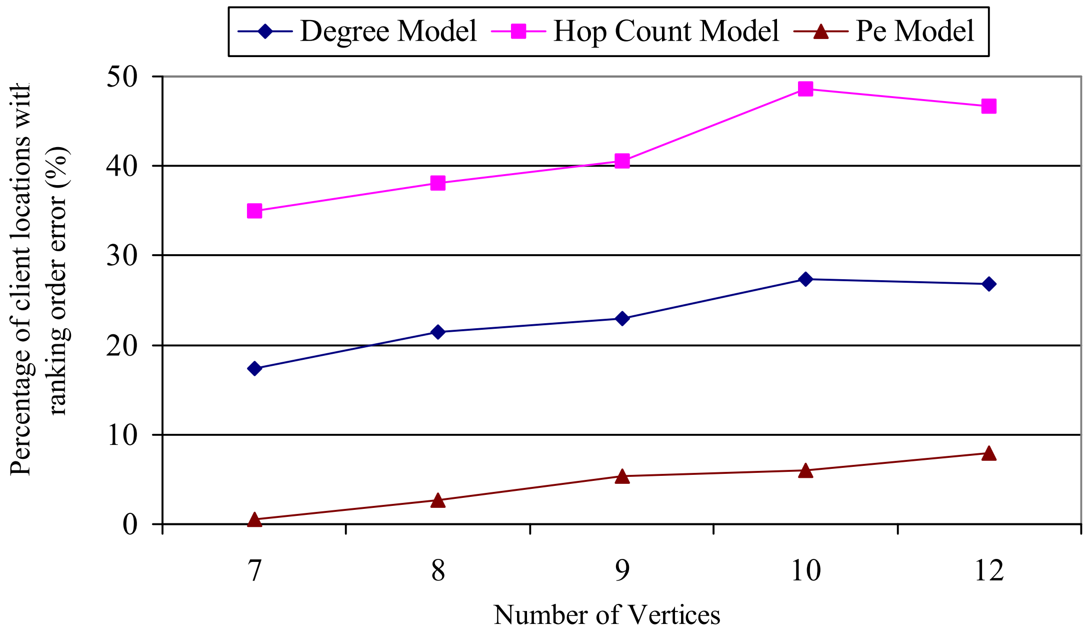

Figure 25. Ranking Analysis Graphical Results (Frequency of Errors)

Table 10 and Figure 26 present the average magnitude of errors per error occurrence. For each occurrence of error, the P_e model has an average error magnitude of less than 2. This shows that even if the P_e model differs from the Min-cut reference, the difference is minimal. This again indicates that the P_e model is more accurate than the other heuristic models.

Number of Vertices	**Ranking Analysis (Average magnitude of error per ranking error occurrence)**		
	Degree Model	Hop Count Model	P_e Model
7	2.0492	3.6571	1.5
8	3.5640	4.4557	1.7619
9	4.8502	4.3616	1.8542
10	5.6642	5.5165	1.9333
12	8.3758	7.8982	1.9053

Table 10. Average Error Magnitude Per Error Occurrence

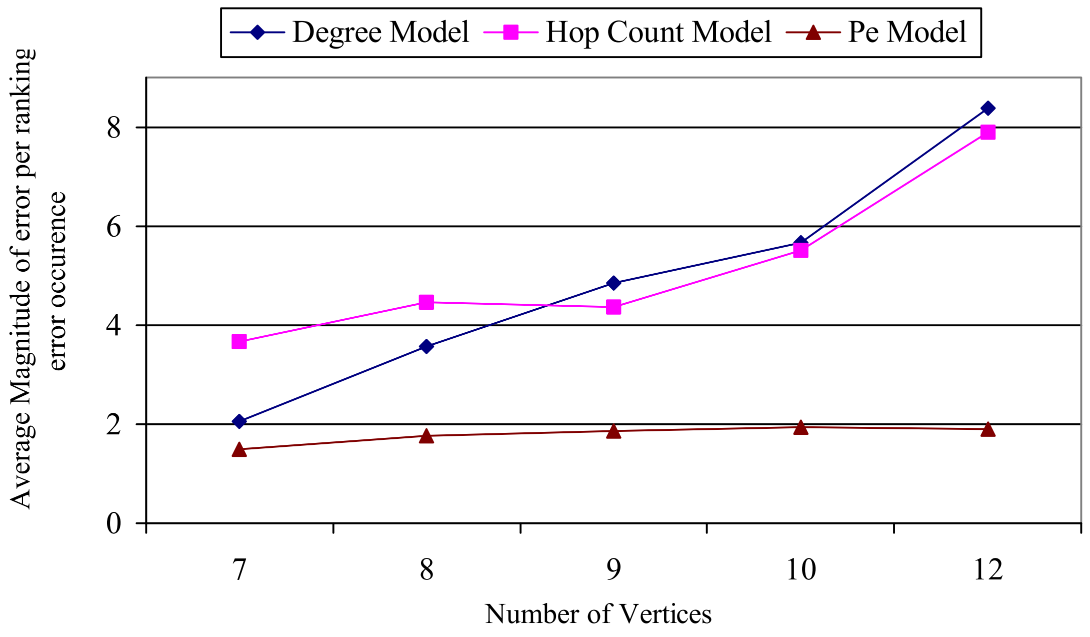

Figure 26. Average Error Magnitude Per Error Occurrence

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION AND FUTURE WORK

A. SYNOPSIS AND CONCLUSION

This thesis's goal was to find an efficient and effective method to rate the suitability of a node to host a critical service like the SAAM server application. The locality of the server node is very critical to its clients. Thus, the selection of the server-hosting node was taken from the client's point of view. The server must be on a node where the connection to the client is the most survivable. A general metric was defined to measure the connection survivability of each of the potential server hosts.

However, the computation of the metric involves an exponential complexity algorithm, which is time consuming, especially when the network size is large. Three polynomial complexity heuristic models have been developed to compute the best choice of placing the server on a node instead. The first model is based on the degree of the potential node, the second one is based on proximity of the node, and the third one, called the P_e model, is based on the number of critical edges of the connection between the potential node and the client.

A set of experiments has been conducted to compare the accuracy of the three heuristic models. The results showed that the P_e model is most accurate. The P_e model incurred almost 0% of errors in determining the best node. A more thorough analysis indicates that the P_e model has a 92% of accuracy in ranking of all nodes for a network size of up to 12 nodes. An on the average, an error occurrence with the P_e model caused a node to be out of order by only two positions.

B. FUTURE WORK

Several potential tasks are left as areas of future work. The following sections briefly touch on some of these areas. However, this is not a comprehensive list.

1. Comprehensive Testing of Heuristic Model Based on Critical Nodes Computation

The experiment conducted in this thesis only involves network sizes of up to 12 routers. As the SAAM system has the capability to handle up to 40 routers, more simulations and analyses need to be carried out on larger network topologies. This is to verify that the algorithm gives consistent results on larger networks.

2. Computation of the Optimal Node Based on Different Client Weighting System

The current computation of the optimum node for hosting the SAAM server is based on one individual client. However, the optimum choice of the hosting node must take into consideration of all clients, which may have different levels of needs from the server. There must be a way to solve this issue. One feasible solution is to use a weighting system to combine the rankings based on different clients into one metric. The more important clients may be assigned heavier weights.

3. Incorporating Heuristic Model into SAAM Server Agent

In order for the SAAM server to have the intelligence to determine the optimum host, the ranking algorithm developed by this thesis needs to be incorporated into the SAAM server agent. Future thesis work may want to look into an efficient way of integrating the algorithm into the SAAM system, which already supports limited server mobility.

APPENDIX A. SIMULATION PROGRAM SOURCE CODES

```
/*
 * @(#)AnalyzeDataFile.java
 *
 */
package tw.edu.ncnu.im.cnclab.JGAP.UI;

import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Algor.*;
import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;
import java.io.*;
import java.util.*;

/**
 *  The AnalyzeDataFile class reads in the simulation result file to be analyze
 *  by the algorithms.
 *
 *  @version 0.1  Mar 1, 2003
 *  @author Eng Hong Chua
 */
class AnalyzeDataFile
{
   private String fileToAnalyze;
   private String resultFile;
   private String strResult ="";

   private int[] aryKe;
   private int[] aryCutSet;
   private int[] aryDegree;
   private int[] aryHopCount;
   private double[] aryPe;

   private int noOfVertices = 0;
   private int noOfGraphs=0;

   private int intDegreeError = 0;
   private int intHopCountError = 0;
   private int intPeError = 0;

   private int intDegreeMagnitudeError = 0;
   private int intHopCountMagnitudeError = 0;
   private int intPeMagnitudeError = 0;

   private int intNoOfBestNodeDegreeError = 0;
   private int intNoOfBestNodeHopCountError = 0;
   private int intNoOfBestNodePeError = 0;

   private int intDegreeFirstRankNodes = 0;
   private int intHopCountFirstRankNodes = 0;
   private int intPeFirstRankNodes = 0;

   /**
    * Constructor of the class
    */
   public AnalyzeDataFile(String dataFilename, String resultFilename)
   {
      fileToAnalyze = dataFilename;
      resultFile = resultFilename;
   }

   /**
    * This methods analyzes the data read from the simulation data files by calling
    * the respective Analysis Algorithm (DataAnalysisBestNode and
DataAnalysisErrorNodeRemoval).
    * It then reads the analysis results and save them into a external file
    * @return boolean
    */
   public boolean analyzeData()
   {
      FileReader fileReader;
      BufferedReader br;
      String inputString;
      String stringData;
      StringTokenizer st;
      boolean dataComing;
      int row=0;
      String dataToken;
      dataComing = false;

      try
      {
         fileReader = new FileReader(fileToAnalyze);
         br = new BufferedReader(fileReader);
         while((inputString = br.readLine()) != null)
         {
            System.out.println(inputString);
            if (inputString.length() == 0)
            {
               writeToResultFile(resultFile, "\n");
            }

            if (inputString.length() > 0)
            {
               if (inputString.charAt(0) != '#')
               {
                  writeToResultFile(resultFile, inputString + "\n");
               }
               if (inputString.charAt(0) == '*')
               {
                  System.out.println(inputString);
               }
               if (inputString.charAt(0) == '@')
               {
                  st = new StringTokenizer(inputString, ",");
                  while (st.hasMoreTokens())
                  {
                     String info = st.nextToken();
                     if (info.charAt(0) == '@')
                     {
                        info = info.substring(1);
                        System.out.println(info);
                        noOfGraphs = (new Integer(info)).intValue();
                     }
                     else
                     {
                        System.out.println(info);
                        noOfVertices = (new Integer(info)).intValue();
                     }
                  }
               }
               else if (inputString.charAt(0) == '!')
               {
                  System.out.println("found starter string");
                  dataComing = true;
                  row = 0;
                  aryKe = new int[noOfVertices-1];
                  aryCutSet = new int[noOfVertices-1];
                  aryDegree = new int[noOfVertices-1];
                  aryHopCount = new int[noOfVertices-1];
                  aryPe= new double[noOfVertices-1];
               }
               else if (inputString.charAt(0) != '#' && inputString.charAt(0) != '@' &&
inputString.charAt(0) != '%')
```

```
          {
             if (dataComing)
             {
                st = new StringTokenizer(inputString, ";");

                while (st.hasMoreTokens())
                {
                   String dataString = st.nextToken();
                   StringTokenizer dt = new StringTokenizer(dataString, ",");
                   int positionCount = 0;
                   while (dt.hasMoreTokens())
                   {
                      stringData = dt.nextToken();
                      switch (positionCount)
                      {
                         case 0:  aryKe[row] = (new Integer(stringData)).intValue();
                                  break;
                         case 1:  aryCutSet[row] = (new
Integer(stringData)).intValue();
                                  break;
                         case 2:  aryDegree[row] = (new
Integer(stringData)).intValue();
                                  break;
                         case 3:  aryHopCount[row] = (new
Integer(stringData)).intValue();
                                  break;
                         case 4:  aryPe[row] = (new
Double(stringData)).doubleValue();
                                  break;
                         default: System.out.println("Error in data conversion!");
                      }  // end switch

                      positionCount++;
                      if (positionCount == 5)
                         break;
                   } // end while
                }  //end while

             }  // end if
             row++;
             if (row == noOfVertices -1 && dataComing)
             {
                dataComing = false;

                // Analysis the data
                DataAnalysisBestNode analysisBestNode = new
DataAnalysisBestNode(aryKe, aryCutSet, aryDegree, aryHopCount, aryPe);
                analysisBestNode.analyzeDataBasedOnBestNode();
                writeToResultFile(resultFile, analysisBestNode.getStringResult());

                // Tracking no. of errors
                intNoOfBestNodeDegreeError += analysisBestNode.getNoOfDegreeError();
                intNoOfBestNodeHopCountError +=
analysisBestNode.getNoOfHopCountError();
                intNoOfBestNodePeError += analysisBestNode.getNoOfPeError();

                intDegreeFirstRankNodes +=
analysisBestNode.getNoOfDegreeFirstRankNodes();
                intHopCountFirstRankNodes +=
analysisBestNode.getNoOfHopCountFirstRankNodes();
                intPeFirstRankNodes += analysisBestNode.getNoOfPeFirstRankNodes();

                // Ranking based on Error Node removal
                DataAnalysisErrorNodeRemoval analysisNodeRemoval = new
DataAnalysisErrorNodeRemoval(aryKe, aryCutSet, aryDegree, aryHopCount, aryPe);
                analysisNodeRemoval.analyzeDataBasedOnErrorNodeRemoval();
                writeToResultFile(resultFile,
analysisNodeRemoval.getStringResult());
                writeToResultFile(resultFile, "\n");

                if (analysisNodeRemoval.getDegreeErrorFound())
```

```
                    {
                       intDegreeError++;
                    }
                    if (analysisNodeRemoval.getHopCountErrorFound())
                    {
                       intHopCountError++;
                    }
                    if (analysisNodeRemoval.getPeErrorFound())
                    {
                       intPeError++;
                    }

                    intDegreeMagnitudeError
+=analysisNodeRemoval.getDegreeMagnitudeErrorPerGraph();
                    intHopCountMagnitudeError
+=analysisNodeRemoval.getHopCountMagnitudeErrorPerGraph();
                    intPeMagnitudeError
+=analysisNodeRemoval.getPeMagnitudeErrorPerGraph();
                 }
              }
           }
        }
        writeErrorAnalysisResultToFile();
        br.close();
        fileReader.close();
        return true;
     }
     catch (Exception e)
     {
        System.out.println("Error...");
        e.printStackTrace();
        return false;
     }
  }

  /**
   * This methods writes the analysis results to an external file.
   */
  private void writeErrorAnalysisResultToFile()
  {
     double dbAverageDegreeError = 0;
     double dbAverageHopCountError = 0;
     double dbAveragePeError = 0;

     // Compute best Node errors
     writeToResultFile(resultFile, "## Best Node Analysis\n" );
     writeToResultFile(resultFile, "# Total Degree Error=" + intNoOfBestNodeDegreeError
+ "\n");
     writeToResultFile(resultFile, "# Total HopCount Error=" +
intNoOfBestNodeHopCountError + "\n");
     writeToResultFile(resultFile, "# Total Pe Error=" + intNoOfBestNodePeError +
"\n\n");

     writeToResultFile(resultFile, "# Percentage Degree Error=" + (double)
intNoOfBestNodeDegreeError/ intDegreeFirstRankNodes+ "\n");
     writeToResultFile(resultFile, "# Percentage HopCount Error=" +
(double)intNoOfBestNodeHopCountError/intHopCountFirstRankNodes + "\n");
     writeToResultFile(resultFile, "# Percentage Total Pe Error=" + (double)
intNoOfBestNodePeError/intPeFirstRankNodes + "\n\n");

     // Compute Error based on Node removal
     writeToResultFile(resultFile, "## Error Analysis based on Node removal\n" );
     dbAverageDegreeError = (double)intDegreeMagnitudeError / intDegreeError;
     System.out.println("Total Degree Error = " + intDegreeError + ", Average Error = "
+ dbAverageDegreeError);
     writeToResultFile(resultFile, "# Degree Error= " + intDegreeError + ", Average
Error = " + dbAverageDegreeError + "\n");

     dbAverageHopCountError = (double)intHopCountMagnitudeError / intHopCountError;
     System.out.println("Total HopCount Error = " + intHopCountError + ", Average Error
= " + dbAverageHopCountError);
```

```
      writeToResultFile(resultFile, "# HopCount Error= " + intHopCountError + ", Average
Error = " + dbAverageHopCountError + "\n");

      dbAveragePeError = (double)intPeMagnitudeError  / intPeError;
      System.out.println("Total Pe Error = " + intPeError + ", Average Error = " +
dbAveragePeError);
      writeToResultFile(resultFile, "# Pe Error= " + intPeError + ", Average Error = " +
dbAveragePeError + "\n\n");

      System.out.println("Analyzing of Data File Completed!\n");
   }

   /**
    * This methods writes data to an external file.
    * @return boolean
    */
   private boolean writeToResultFile(String strFilename, String strData)
   {
      FileWriter resultFile;

      try
      {
         resultFile = new FileWriter(strFilename, true);
         resultFile.write(strData);
         resultFile.close();
         return true;
      }
      catch (Exception e)
      {
         System.out.print("ERROR encounter in writing");
         e.printStackTrace();
         return false;
      }
   }
}
```

```
/*
 * @(#)DataAnalysisBestNode.java
 *
 */
package tw.edu.ncnu.im.cnclab.JGAP.UI;

import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Algor.*;
import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;
import java.io.*;
import java.util.*;

/**
 *  The DataAnalysisBestNode class analyzes the best nodes in each heuristic results.
 *  This class takes arrays of MinCut, Degree, HopCount and Pe simulation results, and
 *  ranks them and analyzes them using Best Nodes Analysis algorithm.  It has a String
 *  variable which will store the analysis results, which the calling program can read.
 *
 *  @version 0.2  Mar 8, 2003
 *  @author Eng Hong Chua
 */
class DataAnalysisBestNode
{
   private String strResult ="## Analysis on the best node only: \n";

   private int[] aryKe;
   private int[] aryCutSet;
   private int[] aryDegree;
   private int[] aryHopCount;
   private double[] aryPe;

   private int intDegreeErrorCount = 0;
   private int intHopCountErrorCount = 0;
   private int intPeErrorCount = 0;

   private int intDegreeFirstRankNodes = 0;
   private int intHopCountFirstRankNodes = 0;
   private int intPeFirstRankNodes = 0;

   /**
   *  Constructor of the class.
   */
   public DataAnalysisBestNode(int aryKeArray[], int aryCutSetArray[], int
aryDegreeArray[], int aryHopCountArray[], double aryPeArray[])
   {
      setKeArray(aryKeArray);
      setCutSetArray(aryCutSetArray);
      setDegreeArray(aryDegreeArray);
      setHopCountArray(aryHopCountArray);
      setPeArray(aryPeArray);
   }

   /**
   * This method sets the Ke array passed in.
   */
   private void setKeArray(int aryKeArray[])
   {
      aryKe = new int[aryKeArray.length];
      for (int i=0; i<aryKeArray.length; i++)
      {
         aryKe[i] = aryKeArray[i];
      }
   }

   /**
   * This method sets the MinCut array passed in.
   */
```

```
private void setCutSetArray(int aryCutSetArray[])
{
   aryCutSet = new int[aryCutSetArray.length];
   for (int i=0; i<aryCutSetArray.length; i++)
   {
      aryCutSet[i] = aryCutSetArray[i];
   }
}

/**
* This method sets the Degree array passed in.
*/
private void setDegreeArray(int aryDegreeArray[])
{
   aryDegree = new int[aryDegreeArray.length];
   for (int i=0; i<aryDegreeArray.length; i++)
   {
      aryDegree[i] = aryDegreeArray[i];
   }
}

/**
* This method sets the HopCount array passed in.
*/
private void setHopCountArray(int aryHopCountArray[])
{
   aryHopCount = new int[aryHopCountArray.length];
   for (int i=0; i<aryHopCountArray.length; i++)
   {
      aryHopCount[i] = aryHopCountArray[i];
   }
}

/**
* This method sets the Ke array passed in.
*/
private void setPeArray(double aryPeArray[])
{
   aryPe = new double[aryPeArray.length];
   for (int i=0; i<aryPeArray.length; i++)
   {
      aryPe[i] = aryPeArray[i];
   }
}

/**
* This method returns a string of the Analysis results.
* @return string
*/
public String getStringResult()
{
   return strResult;
}

/**
 * This method returns the number of nodes in error in Degree heuristic model.
 * @return int
 */
public int getNoOfDegreeError()
{
   return intDegreeErrorCount;
}

/**
 * This method returns the number of nodes in error in Hop Count heuristic model.
 * @return int
 */
public int getNoOfHopCountError()
{
   return intHopCountErrorCount;
}
```

```
    /**
     * This method returns the number of nodes in error in Hop Count heuristic model.
     * @return int
     */
    public int getNoOfPeError()
    {
       return intPeErrorCount;
    }

    /**
     * This method returns the number of nodes that ranks first in Degree model.
     * @return int
     */
    public int getNoOfDegreeFirstRankNodes()
    {
       return intDegreeFirstRankNodes;
    }

    /**
     * This method returns the number of nodes that ranks first in Hop Count model.
     * @return int
     */
    public int getNoOfHopCountFirstRankNodes()
    {
       return intHopCountFirstRankNodes;
    }

    /**
     * This method returns the number of nodes that ranks first in Pe model.
     * @return int
     */
    public int getNoOfPeFirstRankNodes()
    {
       return intPeFirstRankNodes;
    }

    /**
     * This method analyzes the arrays of heuristic simulation data (Degree, HopCount, Pe)
     * by comparing each set of data with that of the MinCut reference set.  It takes the
     * first ranking group in each set and compares with the first ranking group of the
     * MinCut reference.  If a node is ranked first in the heuristic model, it must be
ranked
     * first in the MinCut reference as well.  If not, it is considered as one error.
     */
    public void analyzeDataBasedOnBestNode()
    {
       boolean found;
       int rank;
       int maxKe;
       int sizeOfGp;
       boolean errorPerClientFound;

       // find max. ke
       maxKe=0;
       for (int i=0; i<aryKe.length; i++)
       {
          if (aryKe[i] > maxKe)
             maxKe = aryKe[i];
       }

       Vector vecHighestKePos = new Vector(aryKe.length);
       // find position (server) with the max Ke
       for (int i=0; i<aryKe.length; i++)
       {
          if (aryKe[i] == maxKe)
             vecHighestKePos.add(new Integer(i));
          else
          {
             // overwrite array with worst result for lower Ke value
             aryCutSet[i] = 1000;
```

```
      aryDegree[i] = 1;
      aryHopCount[i] = 1000;
      aryPe[i] = 1;
   }
}

// find min cut set value
int minCutSet=999;
for (int i=0; i<aryCutSet.length; i++)
{
   if (aryCutSet[i] < minCutSet)
      minCutSet=aryCutSet[i];
}

// Get the lowest cutset into a group and store in a vector
Vector vecLowestCutSetPos = new Vector(aryKe.length);
for (int i=0; i<aryCutSet.length; i++)
{
   if (aryCutSet[i] == minCutSet)
      vecLowestCutSetPos.add(new Integer(i));
}

// find the max degree value
int intHighestDegree=0;
for (int i=0; i<aryDegree.length; i++)
{
   if (aryDegree[i] > intHighestDegree)
      intHighestDegree = aryDegree[i];
}

// find the lowest hop count value
int intLowestHopCount=999;
for (int i=0; i<aryHopCount.length; i++)
{
   if (aryHopCount[i] < intLowestHopCount)
      intLowestHopCount = aryHopCount[i];
}

// find the lowest Pe value
double dbLowestPe=99.0;
for (int i=0; i<aryPe.length; i++)
{
   if (aryPe[i] < dbLowestPe)
      dbLowestPe = aryPe[i];
}

// check that the highest degree node coincide with the lowest cutset node
errorPerClientFound = false;     // use to track error per client per graph
for (int i=0; i<aryDegree.length; i++)
{
   if (aryDegree[i] == intHighestDegree)
   {
      intDegreeFirstRankNodes++;
      found = false;
      for (int j=0; j<vecLowestCutSetPos.size(); j++)
      {
         int tempPos = ((Integer)vecLowestCutSetPos.elementAt(j)).intValue();
         if (tempPos == i)
         {
            found=true;
            break;
         }
      }
      if (!found)
      {
         strResult += "# Wrong estimation for Degree at location=" + i +"\n";
         System.out.println("# Wrong estimation for Degree at location=" + i);
         errorPerClientFound = true;
         intDegreeErrorCount++;
      }
   }
```

```
      }

      errorPerClientFound = false;     // use to track error per client per graph
      for (int i=0; i<aryHopCount.length; i++)
      {
         if (aryHopCount[i] == intLowestHopCount)
         {
            intHopCountFirstRankNodes++;
            found = false;
            for (int j=0; j<vecLowestCutSetPos.size(); j++)
            {
               int tempPos = ((Integer)vecLowestCutSetPos.elementAt(j)).intValue();
               if (tempPos == i)
               {
                  found=true;
                  break;
               }
            }
            if (!found)
            {
               strResult += "# Wrong estimation for HopCount at location=" + i+"\n";
               System.out.println("# Wrong estimation for HopcCount at location=" + i);
               errorPerClientFound = true;
               intHopCountErrorCount++;
            }
         }
      }

      errorPerClientFound = false;     // use to track error per client per graph
      for (int i=0; i<aryPe.length; i++)
      {
         if (aryPe[i] == dbLowestPe)
         {
            intPeFirstRankNodes++;
            found = false;
            for (int j=0; j<vecLowestCutSetPos.size(); j++)
            {
               int tempPos = ((Integer)vecLowestCutSetPos.elementAt(j)).intValue();
               if (tempPos == i)
               {
                  found=true;
                  break;
               }
            }
            if (!found)
            {
               strResult += "# Wrong estimation for Pe at location=" + i +"\n";
               System.out.println("# Wrong estimation for Pe at location=" + i);
               errorPerClientFound = true;
               intPeErrorCount++;
            }
         }
      }

      strResult += "\n";
      System.out.println("ke,cutset,degree,hopcount,pe");
      for (int x=0; x<aryKe.length; x++)
      {
         System.out.println(aryKe[x] + "," + aryCutSet[x] + "," + aryDegree[x] + "," +
aryHopCount[x] + "," + aryPe[x]);
      }
   }
}
```

```
/*
 * @(#)DataAnaylsisErrorNodeRemoval.java
 *
 */
package tw.edu.ncnu.im.cnclab.JGAP.UI;

import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Algor.*;
import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;
import java.io.*;
import java.util.*;

/**
 *  The DataAnalysisErrorNodeRemoval class analyzes the ranking of each of the nodes
 *  in each heuristic model.
 *
 *  @version 0.1  Mar 7, 2003
 *  @author Eng Hong Chua
 */
class DataAnalysisErrorNodeRemoval
{
   private String strResult ="## Analysis based on removal of Error Node and comparing
ranking: \n";
   private int[] aryKe;
   private int[] aryCutSet;
   private int[] aryCutSetDup;
   private int[] aryDegree;
   private int[] aryHopCount;
   private double[] aryPe;

   private int degreeMagnitudeError=0;
   private int hopCountMagnitudeError = 0;
   private int peMagnitudeError = 0;

   private int degreeError = 0;
   private int hopCountError = 0;
   private int peError = 0;

   private boolean degreeErrorFound = false;
   private boolean hopCountErrorFound = false;
   private boolean peErrorFound = false;

   private int intMaxDegreeErrorDiff = 0;
   private int intMaxHopCountErrorDiff = 0;
   private int intMaxPeErrorDiff = 0;

   /**
    * Constructor of the class
    */
   public DataAnalysisErrorNodeRemoval(int aryKeArray[], int aryCutSetArray[], int
aryDegreeArray[], int aryHopCountArray[], double aryPeArray[])
   {
      setKeArray(aryKeArray);
      setCutSetArray(aryCutSetArray);
      setDegreeArray(aryDegreeArray);
      setHopCountArray(aryHopCountArray);
      setPeArray(aryPeArray);
   }

   /**
   * This method sets the Ke array passed in.
   */
   private void setKeArray(int aryKeArray[])
   {
      aryKe = new int[aryKeArray.length];
      for (int i=0; i<aryKeArray.length; i++)
      {
```

```
      aryKe[i] = aryKeArray[i];
   }
}

/**
* This method sets the MinCut array passed in.
*/
private void setCutSetArray(int aryCutSetArray[])
{
   aryCutSet = new int[aryCutSetArray.length];
   for (int i=0; i<aryCutSetArray.length; i++)
   {
      aryCutSet[i] = aryCutSetArray[i];
   }
}

/**
* This method sets the Degree array passed in.
*/
private void setDegreeArray(int aryDegreeArray[])
{
   aryDegree = new int[aryDegreeArray.length];
   for (int i=0; i<aryDegreeArray.length; i++)
   {
      aryDegree[i] = aryDegreeArray[i];
   }
}

/**
* This method sets the HopCount array passed in.
*/
private void setHopCountArray(int aryHopCountArray[])
{
   aryHopCount = new int[aryHopCountArray.length];
   for (int i=0; i<aryHopCountArray.length; i++)
   {
      aryHopCount[i] = aryHopCountArray[i];
   }
}

/**
* This method sets the Pe array passed in.
*/
private void setPeArray(double aryPeArray[])
{
   aryPe = new double[aryPeArray.length];
   for (int i=0; i<aryPeArray.length; i++)
   {
      aryPe[i] = aryPeArray[i];
   }
}

/**
* This method returns a string of the Analysis results.
* @return string
*/
public String getStringResult()
{
   return strResult;
}

/**
 * This method returns the number of error occurence in Degree heuristic model.
 * @return int
 */
public int getNoOfDegreeError()
{
   return degreeError;
}

/**
```

```
     * This method returns the number of error occurence in HopCount heuristic model.
     * @return int
     */
    public int getNoOfHopCountError()
    {
       return hopCountError;
    }

    /**
     * This method returns the number of error occurence in Pe heuristic model.
     * @return int
     */
    public int getNoOfPeError()
    {
       return peError;
    }

    /**
     * This method returns the total magnitude of error in all error occurence
     * in Degree heuristic model.
     * @return int
     */
    public int getDegreeMagnitudeErrorPerGraph()
    {
       return degreeMagnitudeError;
    }

    /**
     * This method returns the total magnitude of error in all error occurence
     * in HopCount heuristic model.
     * @return int
     */
    public int getHopCountMagnitudeErrorPerGraph()
    {

       return hopCountMagnitudeError;
    }

    /**
     * This method returns the total magnitude of error in all error occurence
     * in Pe heuristic model.
     * @return int
     */
    public int getPeMagnitudeErrorPerGraph()
    {
       return peMagnitudeError;
    }

    /**
     * This method returns a boolean indicating whether any error found in Degree
heuristic
     * @return boolean
     */
    public boolean getDegreeErrorFound()
    {
       return degreeErrorFound;
    }

    /**
     * This method returns a boolean indicating whether any error found in HopCount
heuristic
     * @return boolean
     */
    public boolean getHopCountErrorFound()
    {
       return hopCountErrorFound;
    }

    /**
     * This method returns a boolean indicating whether any error found in Pe heuristic
     * @return boolean
     */
    public boolean getPeErrorFound()
    {
       return peErrorFound;
    }

    /**
     * This methods analyzes each node in the heuristic model (Degree, HopCount, Pe)
against
     * that of the MinCut reference set to check for any ranking errors.  A node must be
ranked
     * equal or prior to the same node in the MinCut reference model.  If not, it is
counted as
     * an error, and the Magnitude of each error will be computed.  This node will be
removed
     * from the list so as to prevent cumulative error effect.
     * The result of the analysis is written to a string of result which can be read from
this
     * class.
     */
    public void analyzeDataBasedOnErrorNodeRemoval()
    {
       boolean found;
       int rank;
       int maxKe;
       int sizeOfGp;
       boolean nodeRemoval;
       int nodeCount;

       int actualCutSetRank;
       int actualDegreeRank;
       int actualHopCountRank;
       int actualPeRank;

       // initialize the variable
       degreeMagnitudeError=0;
       hopCountMagnitudeError=0;
       peMagnitudeError=0;

       degreeError = 0;
       hopCountError = 0;
       peError = 0;

       int checkCount;

       // find max ke
       maxKe=0;
       for (int i=0; i<aryKe.length; i++)
       {
          if (aryKe[i] > maxKe)
             maxKe = aryKe[i];
       }

       Vector vecKeGp = new Vector(aryKe.length/2);

       int count = 0;
       int keValue = maxKe;
       do
       {
          Vector vecKePos = new Vector(aryKe.length);
          // find position (server) with the max Ke
          for (int i=0; i<aryKe.length; i++)
          {
             if (aryKe[i] == keValue)
             {
                vecKePos.add(new Integer(i));
                count++;
             }
          }
          if (vecKePos.size() != 0)
          {
             vecKeGp.add(vecKePos);
          }
          keValue--;
       } while (count != aryKe.length && keValue != 0);

       // rank CutSet based on Ke value
       Vector vecCutSetGp = new Vector(aryKe.length);
       sizeOfGp =0;
       for (int i=0; i<vecKeGp.size(); i++)
       {
          count = 0;
          do
          {
             // find lowest CutSet within each group
             int lowestCutSet = 999;
             Vector vecKePos = (Vector)vecKeGp.elementAt(i);
             sizeOfGp = vecKePos.size();
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryCutSet[kePos] < lowestCutSet)
                {
                   lowestCutSet = aryCutSet[kePos];
                }
             }

             // Store the position of lowest Cutset in each grp
             Vector vecCutSetPos = new Vector(vecKePos.size());
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryCutSet[kePos] == lowestCutSet)
                {
                   vecCutSetPos.add(new Integer(kePos));
                   count++;
                   aryCutSet[kePos] = 999;       // rewrite with bogus high value so that
 it will not be considered again
                }
             }

             if (vecCutSetPos.size() !=0)
             {
                vecCutSetGp.add(vecCutSetPos);
             }
          } while (count != sizeOfGp);
       }

       strResult += "# CutSet Ranking:\n";
       for (int i=0; i<vecCutSetGp.size(); i++)
       {
          Vector vecCutSetPos = (Vector)vecCutSetGp.elementAt(i);
          strResult += "# ";
          for (int j=0; j<vecCutSetPos.size(); j++)
          {
             strResult += ((Integer)vecCutSetPos.elementAt(j)).intValue() + ",";
          }
          System.out.println();
          strResult += "\n";
       }

       // rank Degree based on Ke value
       Vector vecDegreeGp = new Vector(aryKe.length);
       sizeOfGp =0;
       for (int i=0; i<vecKeGp.size(); i++)
       {
          count = 0;
          do
          {
             // find lowest CutSet within each group
             int highestDegree = 0;
             Vector vecKePos = (Vector)vecKeGp.elementAt(i);
             sizeOfGp = vecKePos.size();
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryDegree[kePos] > highestDegree)
                {
                   highestDegree = aryDegree[kePos];
                }
             }

             // Store the position of highest Degree in each grp
             Vector vecDegreePos = new Vector(vecKePos.size());
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryDegree[kePos] == highestDegree)
                {
                   vecDegreePos.add(new Integer(kePos));
                   count++;
                   aryDegree[kePos] = 0;       // rewrite with bogus low value so that it
 will not be considered again
                }
             }

             if (vecDegreePos.size() !=0)
             {
                vecDegreeGp.add(vecDegreePos);
             }
          } while (count != sizeOfGp);
       }

       strResult += "# Degree Ranking:\n";
       for (int i=0; i<vecDegreeGp.size(); i++)
       {
          Vector vecDegreePos = (Vector)vecDegreeGp.elementAt(i);
          strResult += "# ";
          for (int j=0; j<vecDegreePos.size(); j++)
          {
             strResult += ((Integer)vecDegreePos.elementAt(j)).intValue() + ",";
          }
          System.out.println();
          strResult += "\n";
       }

       // rank HopCount based on Ke value
       Vector vecHopCountGp = new Vector(aryKe.length);
       sizeOfGp =0;
       for (int i=0; i<vecKeGp.size(); i++)
       {
          count = 0;
          do
          {
             // find lowest HopCount within each group
             int lowestHopCount = 999;
             Vector vecKePos = (Vector)vecKeGp.elementAt(i);
             sizeOfGp = vecKePos.size();
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryHopCount[kePos] < lowestHopCount)
                {
                   lowestHopCount = aryHopCount[kePos];
                }
             }

             // Store the position of lowest Cutset in each grp
             Vector vecHopCountPos = new Vector(vecKePos.size());
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryHopCount[kePos] == lowestHopCount)
```

```
                {
                   vecHopCountPos.add(new Integer(kePos));
                   count++;
                   aryHopCount[kePos] = 999;       // rewrite with bogus high value so
that it will not be considered again
                }
             }

             if (vecHopCountPos.size() !=0)
             {
                vecHopCountGp.add(vecHopCountPos);
             }
          } while (count != sizeOfGp);
       }

       strResult += "# HopCount Ranking: \n";
       for (int i=0; i<vecHopCountGp.size(); i++)
       {
          strResult += "# ";
          Vector vecHopCountPos = (Vector)vecHopCountGp.elementAt(i);
          for (int j=0; j<vecHopCountPos.size(); j++)
          {
             strResult += ((Integer)vecHopCountPos.elementAt(j)).intValue() + ",";
          }
          System.out.println();
          strResult += "\n";
       }

       // rank Pe based on Ke value
       Vector vecPeGp = new Vector(aryKe.length);
       sizeOfGp =0;
       for (int i=0; i<vecKeGp.size(); i++)
       {
          count = 0;
          do
          {
             // find lowest Pe within each group
             double lowestPe = 1.0;
             Vector vecKePos = (Vector)vecKeGp.elementAt(i);
             sizeOfGp = vecKePos.size();
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryPe[kePos] < lowestPe)
                {
                   lowestPe = aryPe[kePos];
                }
             }

             // Store the position of lowest Pe in each grp
             Vector vecPePos = new Vector(vecKePos.size());
             for (int j=0; j<vecKePos.size(); j++)
             {
                int kePos = ((Integer) vecKePos.elementAt(j)).intValue();
                if (aryPe[kePos] == lowestPe)
                {
                   vecPePos.add(new Integer(kePos));
                   count++;
                   aryPe[kePos] = 1.0;       // rewrite with bogus high value so that it
will not be considered again
                }
             }

             if (vecPePos.size() !=0)
             {
                vecPeGp.add(vecPePos);
             }
          } while (count != sizeOfGp);
       }

       strResult += "# Pe Ranking:\n";
```

```
      for (int i=0; i<vecPeGp.size(); i++)
      {
         strResult += "# ";
         Vector vecPePos = (Vector)vecPeGp.elementAt(i);
         for (int j=0; j<vecPePos.size(); j++)
         {
            strResult += ((Integer)vecPePos.elementAt(j)).intValue() + ",";
         }
         strResult += "\n";
      }

      //****** Analysis the ranking follows:

      // ******* For Degree
      // Create a duplicate vecCutSetGp, as the original one may be overwritten in the
process of analysis
      Vector vecCutSetGpDup = duplicateVector(vecCutSetGp);    // cannot use clone()
function alone,
                                                               // 'cos vecCutSetGp is a
vector of vector
      nodeRemoval = false;

      do
      {
         nodeCount = degreeError;
         for (int i=0; i<vecDegreeGp.size(); i++)
         {

            Vector vecDegreePos = (Vector)vecDegreeGp.elementAt(i);
            // compute actual Ranking
            // for instance, Rank1 = 1,3,5; Rank2=2,4; so the actual ranking of 2 and 4
is 4
            actualDegreeRank = 1;
            for (int m=0; m<i; m++)
            {
               Vector vecTempPos = (Vector)vecDegreeGp.elementAt(m);
               actualDegreeRank += vecTempPos.size();
            }
            for (int j=0; j<vecDegreePos.size(); j++)
            {
               int degreePos = ((Integer) vecDegreePos.elementAt(j)).intValue();
               found = false;
               for (int p = 0; p<vecCutSetGp.size(); p++)
               {
                  Vector vecCutSetPos = (Vector)vecCutSetGp.elementAt(p);
                  for (int q=0; q<vecCutSetPos.size(); q++)
                  {

                     int cutSetPos = ((Integer) vecCutSetPos.elementAt(q)).intValue();
                     if (cutSetPos == degreePos)
                     {
                        nodeCount++;
                        found = true;
                        actualCutSetRank = 1;
                        for (int m=0; m<p; m++)
                        {
                           Vector vecTempPos = (Vector)vecCutSetGp.elementAt(m);
                           actualCutSetRank += vecTempPos.size();
                        }
                        // check their actual ranking
                        if (actualDegreeRank < actualCutSetRank)
                        {
                           degreeError++;
                           degreeErrorFound = true;
                           degreeMagnitudeError += actualCutSetRank - actualDegreeRank;
                           strResult += "# Error in Pos " + degreePos + "; DegreeRank=" +
actualDegreeRank + " vs CutSetRank=" + actualCutSetRank + ", DegreeMagnitudeError=" +
degreeMagnitudeError + "\n";
                           // removal error CutSet Node
                           vecCutSetPos.removeElementAt(q);
                           nodeRemoval = true;
                           break;
                        }
                        break;
                     }
                  }
                  if (nodeRemoval || found)
                  {
                     break;
                  }
               }
               if (nodeRemoval)
               {
                  // remove the same node from Degree
                  vecDegreePos.removeElementAt(j);
                  break;
               }
            }
            if (nodeRemoval)
            {
               break;
            }
         }  // for loop
         nodeRemoval = false;
      } while (nodeCount < aryKe.length);


      // ***** For HopCount
      // Re-intialize the CutSetVectorGp, as some nodes may have already been removed by
previous computation
      // & check that all its required data are the same
      checkCount = 0;
      vecCutSetGp = duplicateVector(vecCutSetGpDup);  // copy back CutSetVectorGp
      for (int i = 0; i<vecCutSetGp.size(); i++)
      {
         Vector vecTemp = (Vector)vecCutSetGp.elementAt(i);
         for (int j=0; j<vecTemp.size(); j++)
         {
            int temp = ((Integer) vecTemp.elementAt(j)).intValue();
            checkCount++;
            System.out.print(temp + ", ");
         }
         System.out.println();
      }
      if (checkCount != aryKe.length)
      {
         strResult += "# Error in duplicating vectors\n";
         System.out.println("Error in duplicating vectors");

         // create an error to stop program
         Vector vecTemp = (Vector)vecCutSetGp.elementAt(-99);
      }

      nodeRemoval = false;
      do
      {
         nodeCount = hopCountError;
         for (int i=0; i<vecHopCountGp.size(); i++)
         {
            Vector vecHopCountPos = (Vector)vecHopCountGp.elementAt(i);
            // compute actual Ranking
            // for instance, Rank1 = 1,3,5; Rank2=2,4; so the actual ranking of 2 and 4
is 4
            actualHopCountRank = 1;

            for (int m=0; m<i; m++)
            {
               Vector vecTempPos = (Vector)vecHopCountGp.elementAt(m);
               actualHopCountRank += vecTempPos.size();
            }
            for (int j=0; j<vecHopCountPos.size(); j++)
            {
               int hopCountPos = ((Integer) vecHopCountPos.elementAt(j)).intValue();
               found = false;
               for (int p = 0; p<vecCutSetGp.size(); p++)
               {
                  Vector vecCutSetPos = (Vector)vecCutSetGp.elementAt(p);
                  for (int q=0; q<vecCutSetPos.size(); q++)
                  {

                     int cutSetPos = ((Integer) vecCutSetPos.elementAt(q)).intValue();
                     if (cutSetPos == hopCountPos)
                     {
                        nodeCount++;
                        found = true;
                        actualCutSetRank = 1;
                        for (int m=0; m<p; m++)
                        {
                           Vector vecTempPos = (Vector)vecCutSetGp.elementAt(m);
                           actualCutSetRank += vecTempPos.size();
                        }
                        // check their actual ranking
                        if (actualHopCountRank < actualCutSetRank)
                        {
                           hopCountError++;
                           hopCountErrorFound = true;
                           hopCountMagnitudeError += actualCutSetRank -
actualHopCountRank;
                           strResult += "# Error in Pos " + hopCountPos + ";
HopCountRank=" + actualHopCountRank + " vs CutSetRank=" + actualCutSetRank + ",
HopCountMagnitudeError=" + hopCountMagnitudeError +"\n";

                           // removal error CutSet Node
                           vecCutSetPos.removeElementAt(q);
                           nodeRemoval = true;
                           break;
                        }
                        break;
                     }
                  }
                  if (nodeRemoval || found)
                  {
                     break;
                  }
               }
               if (nodeRemoval)
               {
                  // remove the same node from Degree
                  vecHopCountPos.removeElementAt(j);
                  break;
               }
            }
            if (nodeRemoval)
            {
               break;
            }
         }  // for loop
         nodeRemoval = false;
      } while (nodeCount < aryKe.length);

      // ***** For Pe
      // Re-intialize the CutSetVectorGp, as some nodes may have already been removed by
previous computation
      // & check that all its required data are the same
      vecCutSetGp = duplicateVector(vecCutSetGpDup);     // copy back CutSetVectorGp
      checkCount = 0;
      for (int i = 0; i<vecCutSetGp.size(); i++)
      {
         Vector vecTemp = (Vector)vecCutSetGp.elementAt(i);
         for (int j=0; j<vecTemp.size(); j++)
         {
            int temp = ((Integer) vecTemp.elementAt(j)).intValue();
            checkCount++;
            System.out.print(temp + ", ");
         }
         System.out.println();
      }
      if (checkCount != aryKe.length)
      {
         strResult += "# Error in duplicating vectors\n";
         System.out.println("Error in duplicating vectors");

         // create an error to stop program
         Vector vecTemp = (Vector)vecCutSetGp.elementAt(-99);
      }

      nodeRemoval = false;

      do
      {
         nodeCount = peError;
         for (int i=0; i<vecPeGp.size(); i++)
         {

            Vector vecPePos = (Vector)vecPeGp.elementAt(i);
            // compute actual Ranking
            // for instance, Rank1 = 1,3,5; Rank2=2,4; so the actual ranking of 2 and 4
is 4
            actualPeRank = 1;
            for (int m=0; m<i; m++)
            {
               Vector vecTempPos = (Vector)vecPeGp.elementAt(m);
               actualPeRank += vecTempPos.size();
            }
            for (int j=0; j<vecPePos.size(); j++)
            {

               int pePos = ((Integer) vecPePos.elementAt(j)).intValue();
               found = false;
               for (int p = 0; p<vecCutSetGp.size(); p++)
               {
                  Vector vecCutSetPos = (Vector)vecCutSetGp.elementAt(p);
                  for (int q=0; q<vecCutSetPos.size(); q++)
                  {

                     int cutSetPos = ((Integer) vecCutSetPos.elementAt(q)).intValue();
                     if (cutSetPos == pePos)
                     {
                        nodeCount++;
                        found = true;
                        actualCutSetRank = 1;
                        for (int m=0; m<p; m++)
                        {
                           Vector vecTempPos = (Vector)vecCutSetGp.elementAt(m);
                           actualCutSetRank += vecTempPos.size();
                        }
                        // check their actual ranking
                        if (actualPeRank < actualCutSetRank)
                        {
                           peError++;
                           peErrorFound = true;
                           peMagnitudeError += actualCutSetRank - actualPeRank;
                           strResult += "# Error in Pos " + pePos + "; PeRank=" +
actualPeRank + " vs CutSetRank=" + actualCutSetRank + ", PeMagnitudeError=" +
peMagnitudeError + "\n";
                           // removal error CutSet Node
                           vecCutSetPos.removeElementAt(q);
                           nodeRemoval = true;
                           break;
                        }
                        break;
                     }
                  }
                  if (nodeRemoval || found)
                  {
                     break;
                  }
               }
               if (nodeRemoval)
               {
                  // remove the same node from Degree
                  vecPePos.removeElementAt(j);
                  break;
               }
            }
            if (nodeRemoval)
            {
               break;
            }
         }  // for loop
         nodeRemoval = false;
      } while (nodeCount < aryKe.length);

      if (degreeMagnitudeError > 0)
      {
         strResult += "# Degree Error = " + degreeError + ", Ave Error Mag. =" +
(double)degreeMagnitudeError / degreeError + "\n";
      }
      if (hopCountMagnitudeError > 0)
      {
         strResult += "# HopCount Error = " + hopCountError + ", Ave Error Mag. = " +
(double)hopCountMagnitudeError / hopCountError + "\n";
      }
      if (peMagnitudeError > 0)
      {
         strResult += "# Pe Error = " + peError + ", Ave Error Mag. = " +
(double)peMagnitudeError / peError + "\n";
      }
   }

   /**
    * This method copies one Vector into another and returns it to the calling routine.
    * @return Vector
    */
   private Vector duplicateVector(Vector vecInputGp)
   {
      Vector vecOutputGp = new Vector(vecInputGp.size());
      for (int i = 0; i<vecInputGp.size(); i++)
      {
         Vector vecTemp = (Vector)vecInputGp.elementAt(i);
         Vector vecTempCopied = (Vector)vecTemp.clone();

         vecOutputGp.add(vecTempCopied);
      }
      return vecOutputGp;
   }
}
```

```
/*
 * @(#)FindSingleCriticalEdgesPair.java
 *
 */
package tw.edu.ncnu.im.cnclab.Algor;

import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Tools.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.JGAP.UI.*;
import java.util.*;
import java.awt.*;


/**
 *  The FindSingleCriticalPair class is used to find all the critical edges for each pair
 *  of nodes in a graph.  Its objective is to find the number of cutset
 *
 *  @version 0.1  Feb 1, 2003
 *  @author  Baris Aktop
 *  Modified by: Eng Hong Chua
 */
public class FindSingleCriticalEdgesPair extends Algorithm {
   //this Ke will be used to store temp ke values for the graphs that are
   //obtained by subtracting every Ke combination of edges of the main graph
   int tempKe = 0;
   //this Ke will be used to store ke value of the main graph
   int Ke = 0;


   protected int verts;

   /**
    *  Indicate the edges does not containes non-zero flow.
    */
   public static final Color OTHER=Edge.untouchedColor;

   /**
    *  Indicate the edges containes non-zero flow.
    */
   public static final Color FLOW_EDGE=Edge.normalColor;
   /**
    *  The residual graph of <code> graph </code>
    */
   private Graph resG;

  //Vector to store identities of critical edges
   protected Vector Critical;

   private int mintCutSet;

   // return the result for each iteration (each client) to the caller
   public int getCutSet()
   {
      return mintCutSet;
   }

   /**
    * Constructor of algorithm FindCriticalEdges.
    */
   public FindSingleCriticalEdgesPair() {

      legendPrompt=new String[]{"Non-Critical","Critical"};
      legendColor=new Color[]{OTHER,FLOW_EDGE};
      resG = new Graph();
   }

  /**
   * Set arguments from the argument list.
   * <p>
   * Valid args: <code> null </code>, <code> Object[]{Integer(int
<I>sourceNode</I>),Integer(int <I>sinkNode</I>)} </code>
   * @param args        The list of arguments.
   * @exception IllegalArgumentException Illegal argument defined.
   */
   protected void setArg(Object args[]) throws IllegalArgumentException{
      Enumeration enum=graph.verticesElements();
      Vertex v=(Vertex) enum.nextElement();
      startNode=v.getID();
      if (!enum.hasMoreElements()){
         throw new InvalidGraphTypeException("Graph must contain more than 2 vertices.");
      }
      v=(Vertex) enum.nextElement();
      endNode=v.getID();

      if (args==null)
         return;
      if (args.length!=2)
         throw new IllegalArgumentException("Wrong number of arguments");
      switch(args.length){
         case 2:
            try{
               startNode=((Integer) args[0]).intValue();
            }catch(Exception e){
               throw new IllegalArgumentException("Wrong format of arguments[0]");
            }
            try{
               endNode=((Integer) args[1]).intValue();
            }catch(Exception e){
               throw new IllegalArgumentException("Wrong format of arguments[1]");
            }
            break;
      }
   }

   /**
    *  Preferred dialog: Ask2VertexDialog.
    *  @param graph The Graph.
    *  @param fr    The parent Frame.
    *  @return the preferred dialog.
    */
   public GenericDialog getPreferredDialog(Graph graph,Frame fr){
      return new Ask2VertexDialog(graph,fr,"Vertex Dialog","Input the source vertex and
sink vertex?",true);
   }


   private boolean augmentingPathExist(Graph graph){

      resG=(Graph) graph.clone();
      ResidualGraph rg=new ResidualGraph();
      rg.execute(resG,new Object[]{new Integer(startNode),new Integer(endNode)});

      BFS bfs=new BFS();
      bfs.execute(resG,new Object[]{new Integer(startNode),new Boolean(false)});
      if (resG.getPredecessorNode(endNode)==resG.NUL){
         return false;
      }
      return true;
   }

   private void addAugmenting(Graph graph){
      double cf=Double.POSITIVE_INFINITY;
      double d;
      int u,v;
      v=endNode;
      for(u=resG.getPredecessorNode(v);u!=Graph.NUL;
       u=resG.getPredecessorNode(v)){
         d=graph.getEdge(u,v).getWeight()-graph.getEdge(u,v).getFlow();
         if (cf>d)
            cf=d;
         v=u;
      }

      v=endNode;
      for(u=resG.getPredecessorNode(v);u!=Graph.NUL;
       u=resG.getPredecessorNode(v)){
         graph.getEdge(u,v).setFlow(graph.getEdge(u,v).getFlow()+cf);
         v=u;
      }
   }

   private void colorFlow(Graph graph, int ke)
   {
      if (ke == 1)
         {
            for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();)
            {
              Edge edge=(Edge) e.nextElement();

              if ((edge.getKe()) != 0 )
              {
                paintCritical(edge);
              } else
              {
                paintNonCritical(edge);
              }
            }//end for
          }else
          {
            System.out.println("Since Ke is greater than 1, critical edges will not be
painted to red");
          }

      graph.getVertex(startNode).setColor(Color.green);
      graph.getVertex(endNode).setColor(Color.yellow);
   }//end of colorFlow()

   private double flowOut(Graph graph,int node){
      Edge edge;
      double flow=0.0;
      for(Enumeration e=graph.edgesElements(node);e.hasMoreElements();){
         edge=(Edge) e.nextElement();
         flow+=edge.getFlow();
      }
      return flow;
   }



   /**
    * Implementation of this algorithm.
    * @return <code> Double(double <I> flow_of_the_graph</I>) </code>
    */
   protected Object algorImpl()
   {
      Edge edge;
      double flow=0.0;

      Critical = new Vector();
      //you don't need to show flows
      graph.setShowFlow(false);

      System.out.println(startNode + " --> " + endNode);
      for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();){
         edge=(Edge) e.nextElement();
         edge.setFlow(0.0);
      }

      while(augmentingPathExist(graph)){
         addAugmenting(graph);
      }
```

```
   flow=flowOut(graph,startNode);

   Ke = (int)flow;

   outputMessage="Ke = "+StringTool.doubleToString(flow,2) + "    ";
   iterate(graph);

   return new Double(flow);

}//end of algorImpl()

protected void setKe (int ke)
{
   tempKe = ke;
}

protected void paintCritical(Edge e)
{
   Edge edge = e;
   edge.setColor(Color.red);
}

 protected void paintNonCritical(Edge e)
{
   Edge edge = e;
   edge.setColor(Color.blue);
}

protected void iterate (Graph graph)
{
 Graph ana = graph;
 int from = 0;
 int to = 0;
 int numOfEdges = ana.getNumOfEdges();
 Edge temp1 = null;
 Edge temp2 = null;
 Edge temp3 = null;
 Edge temp4 = null;
 Edge edge = null;
 ana.enumerateEdges();
 Graph tempGraph = ana;
 //ShowCriticalEdges criticalEdges = new ShowCriticalEdges();

 switch (Ke)
 {
   case 1:
   //do nothing
   break;//end of case 1

   case 2:
   //we need ke times nested for loops to try every combination of ke edge pairs
   for(int i = 0 ; i < numOfEdges ; i++)
   {
     for (int j = i+1; j < numOfEdges; j++)
     {
       //get the a pair of edges
       temp1 = tempGraph.getEdge(i);
       temp2 =tempGraph.getEdge(j);

       //delete this pair of edges from the graph
       tempGraph.deleteEdge(temp1);
       tempGraph.deleteEdge(temp2);

       //find Ke on this new graph
       tempKe = (int)reFindKe(tempGraph);

       //test if Ke has dropped to zero which means this pair is critical
       if (tempKe == 0)
       {
         Critical.add(temp1);
```

```
      Critical.add(temp2);
    }//end if

    //Add the deleted edges back for next iteration,
    // make sure that they have the same edgeIds as before
    tempGraph.addEdge(temp1, temp1.getEdgeID());
    tempGraph.addEdge(temp2, temp2.getEdgeID());
  }//end for
}//end for
break;//end of case 2

case 3:
//we need ke times nested for loops to try every combination of ke edge pairs
for(int i = 0 ; i < numOfEdges ; i++)
{
  for (int j = i+1; j < numOfEdges; j++)
  {
    for(int k = j+1; k < numOfEdges; k++)
    {
      //get the a pair of edges
      temp1 = tempGraph.getEdge(i);
      temp2 = tempGraph.getEdge(j);
      temp3 = tempGraph.getEdge(k);

      //delete this pair of edges from the graph
      tempGraph.deleteEdge(temp1);
      tempGraph.deleteEdge(temp2);
      tempGraph.deleteEdge(temp3);


      //find Ke on this new graph
      tempKe = (int)reFindKe(tempGraph);

      //test if Ke has dropped to zero which means this pair is critical
     if (tempKe == 0)
     {
        Critical.add(temp1);
        Critical.add(temp2);
        Critical.add(temp3);
      }//end if

      //Add the deleted edges back for next iteration,
      // make sure that they have the same edgeIds as before
      tempGraph.addEdge(temp1, temp1.getEdgeID());
      tempGraph.addEdge(temp2, temp2.getEdgeID());
      tempGraph.addEdge(temp3, temp3.getEdgeID());
    }//end for
  }//end for
}//end for
break;//end of case 3

case 4:
//we need ke times nested for loops to try every combination of ke edge pairs
for(int i = 0 ; i < numOfEdges ; i++)
{
  for (int j = i+1; j < numOfEdges; j++)
  {
    for(int k = j+1; k < numOfEdges; k++)
    {
      for (int n = k+1; n < numOfEdges; n++)
      {
        //get the a pair of edges
        temp1 = tempGraph.getEdge(i);
        temp2 = tempGraph.getEdge(j);
        temp3 = tempGraph.getEdge(k);
        temp4 = tempGraph.getEdge(n);

        //delete this pair of edges from the graph
        tempGraph.deleteEdge(temp1);
        tempGraph.deleteEdge(temp2);
        tempGraph.deleteEdge(temp3);
```

```
         tempGraph.deleteEdge(temp4);

         //find Ke on this new graph
         tempKe = (int)reFindKe(tempGraph);

         //test if Ke has dropped to zero which means this pair is critical
         if (tempKe == 0)
         {
           Critical.add(temp1);
           Critical.add(temp2);
           Critical.add(temp3);
           Critical.add(temp4);
         }//end if

         //Add the deleted edges back for next iteration,
         // make sure that they have the same edgeIds as before
         tempGraph.addEdge(temp1, temp1.getEdgeID());
         tempGraph.addEdge(temp2, temp2.getEdgeID());
         tempGraph.addEdge(temp3, temp3.getEdgeID());
         tempGraph.addEdge(temp4, temp4.getEdgeID());
       }//end for
     }//end for
   }//end for
 }//end for
 break;//end of case 4

 default:
 System.out.println("Ke is too big, no switch case for this Ke");
}//end switch

//Array to identify the critical edges
int [] criticalEdgeIdentities = new int [numOfEdges];
//Initialize the array
for (int m = 0 ; m < criticalEdgeIdentities.length ; m++)
{
   criticalEdgeIdentities [m] = 0;
}
//Print out the critical edge pairs
System.out.println("critical edges are : ");


if (Ke == 1)
{
   for(Enumeration e=tempGraph.allEdgesElements();e.hasMoreElements();)
   {
    edge=(Edge) e.nextElement();
    if((int)edge.getFlow() == 1)
    {
       System.out.println(edge.getEdgeID() + " " );
     }//end if
   }//end for
 }//end if

if (Ke == 2)
{
   for (int k = 0 ; k < Critical.size(); k++)
   {
     Edge e1 = (Edge)Critical.get(k);
     Edge e2 = (Edge)Critical.get(k+1);
     criticalEdgeIdentities [e1.getEdgeID()] = 1;
     criticalEdgeIdentities [e2.getEdgeID()] = 1;
     System.out.println("(" + e1.getEdgeID() + "," + e2.getEdgeID() + ")");
     k= k+1;
   }//end for
 }//end if

if (Ke == 3)
{
   for (int k = 0 ; k < Critical.size(); k++)
   {
     Edge e1 = (Edge)Critical.get(k);
```

```
          Edge e2 = (Edge)Critical.get(k+1);
          Edge e3 = (Edge)Critical.get(k+2);
          criticalEdgeIdentities [e1.getEdgeID()] = 1;
          criticalEdgeIdentities [e2.getEdgeID()] = 1;
          criticalEdgeIdentities [e3.getEdgeID()] = 1;
          System.out.println("(" + e1.getEdgeID() + "," + e2.getEdgeID() + "," +
e3.getEdgeID() + ")");
          k= k+2;
        }//end for
      }//end if

      if (Ke == 4)
      {
        for (int k = 0 ; k < Critical.size(); k++)
        {
          Edge e1 = (Edge)Critical.get(k);
          Edge e2 = (Edge)Critical.get(k+1);
          Edge e3 = (Edge)Critical.get(k+2);
          Edge e4 = (Edge)Critical.get(k+3);
          criticalEdgeIdentities [e1.getEdgeID()] = 1;
          criticalEdgeIdentities [e2.getEdgeID()] = 1;
          criticalEdgeIdentities [e3.getEdgeID()] = 1;
          criticalEdgeIdentities [e4.getEdgeID()] = 1;
          System.out.println("(" + e1.getEdgeID() + "," + e2.getEdgeID() + "," +
e3.getEdgeID() + "," + e4.getEdgeID() + ")");
          k= k+3;
        }//end for
      }//end if
      mintCutSet = (Critical.size()) / Ke;
   }//end iterate()


   public double reFindKe(Graph g)
   {
      Graph changed = g;
      Edge edge;
      double flow = 0;

      for(Enumeration e=changed.allEdgesElements();e.hasMoreElements();){
         edge=(Edge) e.nextElement();
         edge.setFlow(0.0);
      }

      while(augmentingPathExist(changed)){
         addAugmenting(changed);
      }

      flow=flowOut(changed,startNode);

      return flow;

   }//end of reFindKe()

}//end of FindCriticalEdges()
```

```
/*
 * @(#)ProbabilityOfEdgeFailure.java
 *
 */
package tw.edu.ncnu.im.cnclab.Algor;

import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Tools.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.JGAP.UI.*;
import java.util.*;
import java.awt.*;


/**
 *  The ProbabilityOfEdgeFailure class is used to find Probability of Edge Failure
 *  based on FordFulkerson method.
 *
 *  Flow edge can be identified as edges with state = FLOW_EDGE
 *
 *  @version 0.3  Jan 15, 2003
 *  @author  Eng Hong Chua
 */
public class ProbabilityOfEdgeFailure extends Algorithm {
   protected int verts;
   private int counter=0;

   /**
    *  Indicate the edges does not containes non-zero flow.
    */
   public static final Color OTHER=Edge.untouchedColor;

   /**
    *  Indicate the edges containes non-zero flow.
    */
   public static final Color FLOW_EDGE=Edge.normalColor;
   /**
    *  The residual graph of <code> graph </code>
    */
   private Graph resG;

   private Vector pathSequenceVector;

   private String strResult;

   private int mintKe;
   private int mintDegree;
   private int mintTotalHopCount;
   private double mdbPe;

   /**
    * return the result for each iteration (each client) to the caller
    * @return String
    */
   public String getResult()
   {
      return strResult;
   }

   /**
    * Return the value of Ke.
    * @return int
    */
   public int getKe()
   {
      return mintKe;
   }

   /**
    * Return the value of Degree.
```

```
     * @return int
     */
    public int getDegree()
    {
       return mintDegree;
    }

    /**
     * Return the value of HopCount.
     * @return int
     */
    public int getTotalHopCount()
    {
       return mintTotalHopCount;
    }

    /**
     * Return the value of Pe.
     * @return int
     */
    public double getPe()
    {
       return mdbPe;
    }

    /**
     * Constructor of algorithm ProbabilityOfEdgeFailure based on Ford-Fulkerson.
     */
    public ProbabilityOfEdgeFailure() {
       legendPrompt=new String[]{"Other","Flow"};
       legendColor=new Color[]{OTHER,FLOW_EDGE};
       resG=new Graph();

    }

   /**
    * Set arguments from the argument list.
    */
    protected void setArg(Object args[]) throws IllegalArgumentException{
       Enumeration enum=graph.verticesElements();
       Vertex v=(Vertex) enum.nextElement();
       startNode=v.getID();
       if (!enum.hasMoreElements()){
          throw new InvalidGraphTypeException("Graph must contain more than 2 vertices.");
       }
       v=(Vertex) enum.nextElement();
       endNode=v.getID();

       if (args==null)
          return;
       if (args.length!=2)
          throw new IllegalArgumentException("Wrong number of arguments");
       switch(args.length){
          case 2:
             try{
                startNode=((Integer) args[0]).intValue();
             }catch(Exception e){
                throw new IllegalArgumentException("Wrong format of arguments[0]");
             }
             try{
                endNode=((Integer) args[1]).intValue();
             }catch(Exception e){
                throw new IllegalArgumentException("Wrong format of arguments[1]");
             }
             break;

       }
    }

    /**
     *  Preferred dialog: Ask2VertexDialog.
     *  @return the preferred dialog.
     */
    public GenericDialog getPreferredDialog(Graph graph,Frame fr){
       return new Ask2VertexDialog(graph,fr,"Vertex Dialog","Input the source vertex and
sink vertex?",true);
    }

    /**
     * Find the augmenting path in the graph, and return a boolean expression
     * indicating whether one is found
     * @return boolean
     */
    private boolean augmentingPathExist(Graph graph){
       counter++;
       resG=(Graph) graph.clone();
       ResidualGraph rg=new ResidualGraph();
       rg.execute(resG,new Object[]{new Integer(startNode),new Integer(endNode)});

       BFS bfs=new BFS();
       bfs.execute(resG,new Object[]{new Integer(startNode),new Boolean(false)});
       if (resG.getPredecessorNode(endNode)==resG.NUL){
          return false;
       }
       return true;
    }

    /**
     * Add an augmenting path to the graph.
     */
    private void addAugmenting(Graph graph, Vector pathSequence){
       double cf=Double.POSITIVE_INFINITY;
       double d;
       int u,v;
       v=endNode;
       pathSequence.add(new Integer(v));
       for(u=resG.getPredecessorNode(v);u!=Graph.NUL;
        u=resG.getPredecessorNode(v)){
          d=graph.getEdge(u,v).getWeight()-graph.getEdge(u,v).getFlow();
          if (cf>d)
             cf=d;
          v=u;
          pathSequence.add(new Integer (v));
       }

       v=endNode;
       for(u=resG.getPredecessorNode(v);u!=Graph.NUL;
        u=resG.getPredecessorNode(v)){
          graph.getEdge(u,v).setFlow(graph.getEdge(u,v).getFlow()+cf);
          v=u;
       }
    }

    /**
     * Color the flow of the augmenting path found.
     */
    private void colorFlow(Graph graph){
       graph.reset();
       for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();){
          Edge edge=(Edge) e.nextElement();
          if (edge.getFlow()>0.0){
             edge.setColor(FLOW_EDGE);
          } else{
             edge.setColor(OTHER);
          }

       }
       graph.getVertex(startNode).setColor(Color.green);
       graph.getVertex(endNode).setColor(Color.yellow);
    }

    /**
```

```
 * Find the total flow of the graph
 * @return double
 */
private double flowOut(Graph graph,int node){
   Edge edge;
   double flow=0.0;
   for(Enumeration e=graph.edgesElements(node);e.hasMoreElements();){
      edge=(Edge) e.nextElement();
      flow+=edge.getFlow();
   }
   return flow;
}


/**
 * Implementation of this algorithm.
 * @return Double
 */
protected Object algorImpl() {

   Vertex v;
   Edge edge;
   int from;
   int to;
   int node;
   int counter=0;
   int ke= 0;
   double flow=0.0;

   int degree = 0;
   pathSequenceVector = new Vector (4);
   graph.setShowFlow(true);
   for(Enumeration e=graph.allEdgesElements();e.hasMoreElements();){
      edge=(Edge) e.nextElement();
      edge.setFlow(0.0);

      if (edge.getToNode() == startNode || edge.getFromNode() == startNode)
         degree++;
   }

   while (augmentingPathExist(graph)){
      ke++;
      Vector pathSequence = new Vector(20);

      addAugmenting(graph, pathSequence);
      pathSequenceVector.add(pathSequence);
   }

   // color the graph to show the disjoint paths
   colorFlow(graph);
   flow=flowOut(graph, startNode);

   // create a temp graph of unused edges.
   resG = (Graph) graph.clone();
   resG.deleteAllOtherEdges(OTHER);

   int criticalEdgesProduct = 1;
   int totalHopCount = 0;
   for (int i=0; i < pathSequenceVector.size(); i++)
   {
     Vector pathSequence = (Vector) pathSequenceVector.elementAt(i);
     totalHopCount += pathSequence.size() - 1;
     int criticalEdges = pathSequence.size() - 1;
     for (int j=0; j < pathSequence.size(); j++)
     {
         for (int k=pathSequence.size()-1; k >= j; k--)
         {
             int firstNode = ((Integer)pathSequence.elementAt(j)).intValue();
             int lastNode = ((Integer)pathSequence.elementAt(k)).intValue();

             if (isPathExist(resG, firstNode, lastNode) && firstNode != lastNode)
```

```
                {
                    criticalEdges -= (k - j);
                    j = k;
                    break;
                }
            }

        }
        criticalEdgesProduct *= criticalEdges;
      }

      //Compute Pe
      double pe = (double) criticalEdgesProduct;
      for (int i=0; i < pathSequenceVector.size(); i++)
      {
            pe /= (graph.getNumOfEdges() - i);
      }
      pe *= factorial(pathSequenceVector.size());
      System.out.println("Total Number of Edges=" + graph.getNumOfEdges() + "    " + "ke=
" + ke);
      System.out.println("Pe=" + pe + ", Hop count =" + totalHopCount + ",
Degree(server)= " + degree);

      strResult = startNode + "," + endNode + "," + graph.getNumOfEdges() + ";" + ke +
",";
      outputMessage="Maximum flow = "+StringTool.doubleToString(flow,2) + "    " + "ke= "
+ ke + "    Pe= " + pe + "    ";

      mintKe = ke;
      mintDegree = degree;
      mintTotalHopCount = totalHopCount;
      mdbPe = pe;

      return new Double(flow);


   }

   /**
    * Check whether that there is an alternate path between each pair of nodes and
    * returns a boolean expression indicating so.
    * @return boolean
    */
   private boolean isPathExist(Graph graph, int startNode, int endNode)
   {
      BFS bfs = new BFS();
      bfs.execute(graph, new Object[]{new Integer(startNode), new Boolean(false)});
      if (graph.getPredecessorNode(endNode) == graph.NUL)
      {
        return false;
      }
      return true;
   }

   /**
    * Find the factorial of a number and returns to the calling routine.
    * @return int
    */
   private int factorial (int num)
   {
        if (num == 1)
            return 1;
        else
            return num * factorial(num-1);
   }
}
```

```
/*
 * @(#)Simulation.java
 *
 */
package tw.edu.ncnu.im.cnclab.JGAP.UI;

import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import tw.edu.ncnu.im.cnclab.DataStru.*;
import tw.edu.ncnu.im.cnclab.Algor.*;
import java.awt.*;
import java.awt.event.*;
import java.util.EventObject;
import java.io.*;
import java.util.*;

/**
 *  The Simulation class generates random graphs for the heuristic models and computes
data
 *  required by the reference MinCut set, and heuristic set.  For each graph, each node
is computed
 *  as a client node each time.  The rest of the nodes will simulate as the server node,
and find
 *  the MinCut, Degree, HopCount and Pe value for each of them and save the results in an
external
 *  file.
 *
 *  @version 0.3  Jan 15, 2003
 *  @author Eng Hong Chua
 */
class Simulation
{
   private int noOfGraph;
   private int noOfVertices;
   private double ke;
   private JGAPFrame parentFrame;
   private String strResultFile;

   private int[] aryKe;
   private int[] aryCutSet;
   private int[] aryDegree;
   private int[] aryHopCount;
   private double[] aryPe;

   private int intNoOfBestNodeDegreeError=0;
   private int intNoOfBestNodeHopCountError=0;
   private int intNoOfBestNodePeError=0;

   private int intDegreeError = 0;
   private int intHopCountError = 0;
   private int intPeError = 0;

   private int intDegreeMagnitudeError = 0;
   private int intHopCountMagnitudeError = 0;
   private int intPeMagnitudeError = 0;

   /**
    * Constructor of the class.
    */
   public Simulation(JGAPFrame frame)
   {
      noOfGraph = 1;
      noOfVertices = 8;
      ke = 3.0;
      parentFrame = frame;
   }

   /**
    * Constructor of the class.
    */
   public Simulation(JGAPFrame frame, int noOfRun, int noOfNodesPerGraph, double keValue)
   {
      noOfGraph = noOfRun;
      noOfVertices = noOfNodesPerGraph;
      ke = keValue;
      parentFrame = frame;
   }

   /**
    * This method computes the String equivalent of the month integer passed in.
    * @return String
    */
   private String getStringMonth(int month)
   {
      switch (month)
      {
         case 0 : return "Jan";
         case 1 : return "Feb";
         case 2 : return "Mar";
         case 3 : return "Apr";
         case 4 : return "May";
         case 5 : return "Jun";
         case 6 : return "Jul";
         case 7 : return "Aug";
         case 8 : return "Sep";
         case 9 : return "Oct";
         case 10 : return "Nov";
         case 11 : return "Dec";
         default : return "Error";
      }
   }

   /**
    * This method randomly generates graphs and computes the ranking of each node based 
on
    * each heuristic model (Degree, Hop Count, Pe).  The results are saved into an 
external
    * file.
    */
   public void runSimulation(boolean runCurrentGraphOnly)
   {
      Object[] clientServerNode;
      String strResultData;
      String strGraphName;
      int counter;

      System.out.println("Simulation starts....");
      Date currentDate = new Date();
      strResultFile = "ResultV" + noOfVertices + "G" + noOfGraph + "D" + 
currentDate.getDate() + getStringMonth(currentDate.getMonth()) + ".dat";
      writeToResultFile(strResultFile, "* " + currentDate.toString() + "\n");
      writeToResultFile(strResultFile, "* No. of Graph=" + noOfGraph + ", No of 
Vertices(per graph)=" + noOfVertices+"\n");
      writeToResultFile(strResultFile, "@" + noOfGraph + ","+noOfVertices +"\n");

      for (int graphNo=0; graphNo < noOfGraph; graphNo++)
      {
         if (!runCurrentGraphOnly)
         {
            System.out.println("Create graph");
            WheelGraphRandomizer r=new 
WheelGraphRandomizer(parentFrame.GC[GraphCanvas.GRAPH].getSize());
            parentFrame.graph.copyFrom(r.generate(noOfVertices, ke, 1.0, 1.0, false, 
false));
            strGraphName = "SimV" + noOfVertices + "D" + currentDate.getDate()+ 
getStringMonth(currentDate.getMonth()) + "_GNo" + graphNo + ".gph";
            parentFrame.jgapMenu.saveGraph(strGraphName);

            System.out.println("saving " + strGraphName);
            writeToResultFile(strResultFile, "* " + strGraphName + "\n");
         }
         ProbabilityOfEdgeFailure probEdgeFailure =new ProbabilityOfEdgeFailure();
         Graph gr = (Graph) parentFrame.graph.clone();

         FindSingleCriticalEdgesPair findSingleCriticalEdgesPair = new
FindSingleCriticalEdgesPair();

         for (int clientNodeNo=0; clientNodeNo < gr.getNumOfVertices(); clientNodeNo++)
         {
            aryKe = new int[gr.getNumOfVertices()-1];
            aryCutSet = new int[gr.getNumOfVertices()-1];
            aryDegree = new int[gr.getNumOfVertices()-1];
            aryHopCount = new int[gr.getNumOfVertices()-1];
            aryPe= new double[gr.getNumOfVertices()-1];
            counter = 0;

            writeToResultFile(strResultFile, "!
startNode,endNode,TotalEdges;ke,Cutset,degree,HopCount,Pe \n");
            for (int serverNodeNo=0; serverNodeNo < gr.getNumOfVertices();
serverNodeNo++)
            {
               clientServerNode = new Object[2];
               clientServerNode[0] = new Integer(serverNodeNo);
               clientServerNode[1] = new Integer(clientNodeNo);
               if (serverNodeNo != clientNodeNo)
               {
                  probEdgeFailure.show(gr, clientServerNode);
                  findSingleCriticalEdgesPair.show(gr, clientServerNode);
                  System.out.println("CutSet=" +
findSingleCriticalEdgesPair.getCutSet());
                  strResultData = probEdgeFailure.getResult() +
findSingleCriticalEdgesPair.getCutSet() + ","
                                 + probEdgeFailure.getDegree() + "," +
probEdgeFailure.getTotalHopCount() + ","
                                 + probEdgeFailure.getPe() + "\n";

                  writeToResultFile(strResultFile, strResultData);

                  // save data for analysis
                  aryKe[counter] = probEdgeFailure.getKe();
                  aryCutSet[counter] = findSingleCriticalEdgesPair.getCutSet();
                  aryDegree[counter] = probEdgeFailure.getDegree();
                  aryHopCount[counter] = probEdgeFailure.getTotalHopCount();
                  aryPe[counter] = probEdgeFailure.getPe();
                  counter++;
               }
            }

            // Analysis the data
            // Analysis based on best node
            DataAnalysisBestNode analysisBestNode = new DataAnalysisBestNode(aryKe,
aryCutSet, aryDegree, aryHopCount, aryPe);
            analysisBestNode.analyzeDataBasedOnBestNode();
            writeToResultFile(strResultFile, analysisBestNode.getStringResult());
            // Tracking no. of errors
            intNoOfBestNodeDegreeError += analysisBestNode.getNoOfDegreeError();
            intNoOfBestNodeHopCountError += analysisBestNode.getNoOfHopCountError();
            intNoOfBestNodePeError += analysisBestNode.getNoOfPeError();

            // Ranking based on Error Node removal
            DataAnalysisErrorNodeRemoval analysisNodeRemoval = new
DataAnalysisErrorNodeRemoval(aryKe, aryCutSet, aryDegree, aryHopCount, aryPe);
            analysisNodeRemoval.analyzeDataBasedOnErrorNodeRemoval();
            writeToResultFile(strResultFile, analysisNodeRemoval.getStringResult());
            writeToResultFile(strResultFile, "\n");

            if (analysisNodeRemoval.getDegreeErrorFound())
            {
               intDegreeError++;
            }
            if (analysisNodeRemoval.getHopCountErrorFound())
            {
               intHopCountError++;
            }
            if (analysisNodeRemoval.getPeErrorFound())
            {
               intPeError++;
            }

            intDegreeMagnitudeError
+=analysisNodeRemoval.getDegreeMagnitudeErrorPerGraph();
            intHopCountMagnitudeError
+=analysisNodeRemoval.getHopCountMagnitudeErrorPerGraph();
            intPeMagnitudeError +=analysisNodeRemoval.getPeMagnitudeErrorPerGraph();

            // display the dataset on screen
            System.out.println("ke,cutset,degree,hopcount,pe");
            for (int x=0; x<aryKe.length; x++)
            {
               System.out.println(aryKe[x] + "," + aryCutSet[x] + "," + aryDegree[x] +
"," + aryHopCount[x] + "," + aryPe[x]);
            }

            System.out.println("End of client #" + clientNodeNo);
         }

         System.out.println("End of graph #" + graphNo);
         writeToResultFile(strResultFile, "\n");

         // Print Error after every 10 sets of graph.
         if ((graphNo+1) % 2 == 0 && (graphNo+1) != noOfGraph)
         {
            writeToResultFile(strResultFile, "# No of Graphs SO FAR = " + (graphNo+1) + "
No of vertices (in each graph)= " + noOfVertices + "\n\n");
            writeErrorAnalysisResultToFile(graphNo + 1);
         }
      }
      writeToResultFile(strResultFile, "# No of Graphs = " + noOfGraph + " No of vertices
(in each graph)= " + noOfVertices + "\n\n");
      writeErrorAnalysisResultToFile(noOfGraph);
      System.out.println("Simulation completed!");
   }

   /**
    * This methods writes the analysis results to an external file.
    */
   private void writeErrorAnalysisResultToFile(int graphSize)
   {
      double dbAverageDegreeError = 0;
      double dbAverageHopCountError = 0;
      double dbAveragePeError = 0;

      // Compute best Node errors
      writeToResultFile(strResultFile, "## Best Node Analysis\n" );
      writeToResultFile(strResultFile, "# Total Degree Error=" +
intNoOfBestNodeDegreeError + "\n");
      writeToResultFile(strResultFile, "# Total HopCount Error=" +
intNoOfBestNodeHopCountError + "\n");
      writeToResultFile(strResultFile, "# Total Pe Error=" + intNoOfBestNodePeError +
"\n\n");

      // Compute Error based on Node removal
      writeToResultFile(strResultFile, "## Error Analysis based on Node removal\n" );
      dbAverageDegreeError = (double)intDegreeMagnitudeError / intDegreeError;
      System.out.println("Total Degree Error = " + intDegreeError + ", Average Error = "
+ dbAverageDegreeError);
      writeToResultFile(strResultFile, "# Degree Error= " + intDegreeError + ", Average
Error = " + dbAverageDegreeError + "\n");

      dbAverageHopCountError = (double)intHopCountMagnitudeError / intHopCountError;
      System.out.println("Total HopCount Error = " + intHopCountError + ", Average Error
= " + dbAverageHopCountError);
```

```
      writeToResultFile(strResultFile, "# HopCount Error= " + intHopCountError + ", 
Average Error = " + dbAverageHopCountError + "\n");

      dbAveragePeError = (double)intPeMagnitudeError  / intPeError;
      System.out.println("Total Pe Error = " + intPeError + ", Average Error = " + 
dbAveragePeError);
      writeToResultFile(strResultFile, "# Pe Error= " + intPeError + ", Average Error = " 
+ dbAveragePeError + "\n\n");
   }

   /**
    * This methods writes data to an external file.
    * @return boolean
    */
   private boolean writeToResultFile(String strFilename, String strData)
   {
      FileWriter resultFile;

      try
      {
         resultFile = new FileWriter(strFilename, true);
         resultFile.write(strData);
         resultFile.close();
         return true;
      }
      catch (Exception e)
      {
         System.out.print("ERROR encounter in writing");
         e.printStackTrace();
         return false;
      }
   }
}
```

```
/*
 * @(#)SimulationDialog.java
 *
 */
package tw.edu.ncnu.im.cnclab.JGAP.UI;

import tw.edu.ncnu.im.cnclab.UI.*;
import tw.edu.ncnu.im.cnclab.JGAP.*;
import java.awt.*;
import java.awt.event.*;

/**
 *  The SimulationDialog class shows the dialog for inputing the number of graphs to
 *  be run, the size of each graph, and the Ke value of each node in the graph
 *
 *  @version 0.2  Jan 15, 2003
 *  @author Eng Hong Chua
 */
class SimulationDialog extends GenericDialog implements ActionListener{
   Panel defPanel;
   Panel boundPanel;
   Panel def2Panel;

   /**
    *  TextField for Number of Graph to run.
    */
   private TextField noOfGraphField;

  /**
    *  TextField for Vertices.
    */
   private TextField noOfVerticesField;

   /**
    *  TextField for density.
    */
   private TextField keField;

   //TextField resultFilenameField;

   private Checkbox isRunCurrentGraphCB;

   private int noOfGraph;
   private int noOfVertices;
   private double ke;
   private String strResultFilename;

   /**
    *  Constructor of the class
    */
   public SimulationDialog(Frame fr, String title, String promptText,boolean isModal){
      super(fr, title, promptText, isModal,40);
   }

   /**
    * Initiate the custom panel(central)Panel.
    */
   protected void setCentralPanel(){
      centralPanel.setLayout(new BorderLayout());
      defPanel=new Panel();
      setDefPanel();
      centralPanel.add("North",defPanel);

      boundPanel=new Panel();
      setBoundPanel();
      centralPanel.add("Center",boundPanel);

      def2Panel=new Panel();
      setDirectPanel();
      centralPanel.add("South",def2Panel);
   }
   /**
    * Initiate the Default Panel.
    */
   void setDefPanel(){
      defPanel.add(new Label("Number of Graph to run:"));
      noOfGraphField=new TextField(5);
      defPanel.add(noOfGraphField);
      noOfGraphField.setText("50");

      defPanel.add(new Label("  Number of Vertices [1-"+Graph.MAX_VERTICES+"]:"));
      noOfVerticesField=new TextField(5);
      defPanel.add(noOfVerticesField);
      noOfVerticesField.setText("8");
   }

   /**
    * Sets the default value of each field
    */
   void setBoundPanel()
   {
      boundPanel.add(new Label(" ke:[1-4]:"));
      keField=new TextField(8);
      boundPanel.add(keField);
      keField.setText("3.0");
   }

   protected void setDirectPanel()
   {
      isRunCurrentGraphCB=new Checkbox("Run on Current Graph",false);
      def2Panel.add(isRunCurrentGraphCB);
   }


   /**
    *  Check if the input format valid.
    *  @return boolean
    */
   protected boolean checkFormat(){
      Integer I;
      Double D;

      if ((I=getIntegerFieldValue(noOfGraphField,"No of Graph",1,5000))==null){
         return false;
      }
      noOfGraph=I.intValue();

      if ((I=getIntegerFieldValue(noOfVerticesField,"No. of
Vertices",1,Graph.MAX_VERTICES))==null)
      {
         return false;
      }
      noOfVertices=I.intValue();

      if ((D=getDoubleFieldValue(keField,"ke",1.0,4.0))==null){
         return false;
      }
      ke=D.doubleValue();


      //ke cannot be >= number of vertices
      if ((int)getKe() >= getNoOfVertices())
      {
        msgField.setText("ke cannot be greater than or equal to verts");
        return false;
      }

      return true;
   }
```

```
/**
 * Get the probability of edge appear between vertices.
 *
 * @return int
 */
public int getNoOfGraph()
{
   return noOfGraph;
}

public int getNoOfVertices()
{
      return noOfVertices;
}

public double getKe()
{
      return ke;
}

public String getResultFilename()
{
      return strResultFilename;
}


/**
 * Return whether the simulation is run on the current graph.
 * @return boolean.
 */
public boolean getIsRunCurrentGraph()
{
   return isRunCurrentGraphCB.getState();
}


/**
 * Return whether the graph self-looped .
 * @return whether the graph self-looped .
 */
/*public boolean isSelfLooped(){
   return isSelfLoopedCB.getState();
}*/

/**
 * Return the parameter that the dialog get.
 *  @return The parameters of the dialog get. (Default: <code> additionalArg
 *   <ul>
 *     <ui> Array[0]:  the Integer(vertices)
 *     <ui> Array[1]:  the Double(density)
 *     <ui> Array[2]:  the Double(lowerBound)
 *     <ui> Array[3]:  the Double(upperBound)
 *     <ui> Array[4]:  the Boolean(directed)
 *     <ui> Array[5]:  the Boolean(selfLooped)
 *     <ui> The Other  the argument from the parameter additionalArg.
 *  </ul>
 */
public Object[] getArg(Object[] additionalArg){
   Object[] oArray;
   if (additionalArg!=null){
      oArray=new Object[additionalArg.length+4];
      for(int i=0;i<additionalArg.length;i++){
         oArray[i+4]=additionalArg[i];
      }
   } else{
      oArray=new Object[4];
   }
   oArray[0]=new Integer(noOfGraph);
   oArray[1]=new Integer(noOfVertices);
   oArray[2]=new Double(ke);
   //oArray[3]=new String(strResultFilename);
```

```
        //oArray[3]=new String(upperBound);
        oArray[3]=new Boolean(isRunCurrentGraphCB.getState());
        //oArray[5]=new Boolean(isSelfLoopedCB.getState());
        return oArray;
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[AKT02] Baris Aktop, Ekrem Serin, Selcuk Ozturk, Project 3, June 2002.

[ALD00] Joan M. Aldous, Robin J.Wilson, Graphs and Applications : An Introductory Approach, The Open University, 2000 (ISBN: 1-85233-259-X).

[CAM02] Craig W. Cameron, Steven H. Low, David X. Wei, High-Density Model for Server Allocation and Placement, ACM SIGMETRICS Performance Evaluation Review, Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems, June 2002, Volume 30 Issue 1.

[DARPA98] DARPA, "DARPA/ITO Information Survivability Website", Defense Advanced Research Projects Agency – Information Technology Office, www.darpa.mil/ito/research/is, 1998.

[ELL99] Robert J. Ellison, David A. Fisher, Richard C. Linger, Howard F. Lipson, Thomas A. Longstaff, Nancy R. Mead, Survivability: Protecting Your Critical Systems, IEEE Internet Computing, November/December 1999 (Vol 3, No. 6).

[GIB85] Alan Gibbons, Algorithmic Graph Theory, Cambridge University Press, 1985 (ISBN:0-521-28881-9).

[IEEE90] Institute of Electrical and Electronic Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

[IRS02] IEEE Reliability Society. Reliability Engineering, http://www.ewh.ieee.org/soc/rs/Reliability_Engineering/index.html, Date : January 22, 2003.

[JHA01] Somesh Jha, Jeannette M. Wing, Survivability Analysis of Networked System, Proceedings of the 23rd International Conference of Software Engineering, May 12-19, 2001.

[KAT99] Kati, Efraim, Fault-Tolerant Approach for Deploying Server Agent-based Active network Management (SAAM) server in Windows NT Environment to Provide Uninterrupted Services to Routers In Case of Server Failure(s), Naval Postgraduate School Thesis, March 2000.

[KAR01] Roger Karrer, Thomas R. Gross, Location Selection for Active Services, Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC –10), August 2001.

[LIN98] Richard C. Linger, Nancy R. Mead, Howard F. Lipson, Requirements Definition for Survivable Network Systems, Proceedings of the 1998 International Conference on Requirements Engineering (ICRE '98).

[LIP00] Howard F. Lipson, Survivability – A New Security Paradigm for Protecting Highly Distributed Mission-Critical Systems, IFIP WG10.4 Summer 2000 Meeting, 28 June – 2 July, http://www.dependability.org/wg10.4/meeting38/05-Lipso.pdf, Date: February 14, 2003

[MAR01] Scott Margulis, MAGMA: A Liquid Software Approach to Fault Tolerance, Computer Network Security, and Survivable Networking, Naval Postgraduate School Thesis, December 2001.

[POW02] Richard Power, 2002 CSI/FBI Computer Crime and Security Survey, Computer Security Issues & Trends, Vol. VIII, No. 1, Spring 2002.

[SKI97] Steven S. Skiena, The Algorithm Design Manual, Telos/Springer-Verlag, http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK4/NODE167.HTM, Date: March 6, 2003.

[UMA01] Amjad Umar, Farooq Anjum, Abhrajit Ghosh, Rabih Zbib, Intrusion Tolerant Middleware, DARPA Information Survivability Conference and Exposition II, 2001, DISCEX '01, Proceedings, Volume 2, 2001.

[WEL99] David Wells, Steve Ford, David Langworthy, Nancy Wells, Software Survivability, DARPA Information Survivability Conference and Exposition, 2000, DISCEX '00, Proceedings, Volume 2, 1999.

[WUT99] C. Thomas Wu, An Introduction to Object-Oriented Programming with Java, McGraw Hill, 1999. (ISBN: 0-07-239684-9).

[XIE02] Geoffrey G. Xie, CY03, Homeland Security Research & Technology Research Proposal: "Critical Infrastructure Protection: Maximize Survivability of a Network Dependent Service", 2002.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code CS
   Computer Science Department
   Naval Postgraduate School
   Monterey, California

4. Dr. Geoffrey Xie
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California

5. Dr. Bert Lundy
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California

6. Defence Science & Technology Agency
   Singapore

7. Lt. Baris Aktop
   Turkish Navy
   Turkey

8. Eng Hong Chua
   Singapore